Version Control with Subversion

For Subversion 1.2

(book compiled from Revision 2147)

Ben Collins-Sussman
Brian W. Fitzpatrick
C. Michael Pilato

Version Control with Subversion: For Subversion 1.2: (book compiled from Revision 2147)

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

Published (TBA)

Table of Contents

List of Figures

List of Tables

List of Examples

Foreword

A bad Frequently Asked Questions (FAQ) sheet is one that is composed not of the questions people actually asked, but of the questions the FAQ's author *wished* people had asked. Perhaps you've seen the type before:

Q: How can I use Glorbosoft XYZ to maximize team productivity?

A: Many of our customers want to know how they can maximize productivity through our patented office groupware innovations. The answer is simple: first, click on the “`File`” menu, scroll down to “`Increase Productivity`”, then…

The problem with such FAQs is that they are not, in a literal sense, FAQs at all. No one ever called the tech support line and asked, “How can we maximize productivity?”. Rather, people asked highly specific questions, like, “How can we change the calendaring system to send reminders two days in advance instead of one?” and so on. But it's a lot easier to make up imaginary Frequently Asked Questions than it is to discover the real ones. Compiling a true FAQ sheet requires a sustained, organized effort: over the lifetime of the software, incoming questions must be tracked, responses monitored, and all gathered into a coherent, searchable whole that reflects the collective experience of users in the wild. It calls for the patient, observant attitude of a field naturalist. No grand hypothesizing, no visionary pronouncements here—open eyes and accurate note-taking are what's needed most.

What I love about this book is that it grew out of just such a process, and shows it on every page. It is the direct result of the authors' encounters with users. It began with Ben Collins-Sussman's observation that people were asking the same basic questions over and over on the Subversion mailing lists: What are the standard workflows to use with Subversion? Do branches and tags work the same way as in other version control systems? How can I find out who made a particular change?

Frustrated at seeing the same questions day after day, Ben worked intensely over a month in the summer of 2002 to write *The Subversion Handbook*, a sixty page manual that covered all the basics of using Subversion. The manual made no pretense of being complete, but it was distributed with Subversion and got users over that initial hump in the learning curve. When O'Reilly and Associates decided to publish a full-length Subversion book, the path of least resistance was obvious: just expand the Subversion handbook.

The three co-authors of the new book were thus presented with an unusual opportunity. Officially, their task was to write a book top-down, starting from a table of contents and an initial draft. But they also had access to a steady stream—indeed, an uncontrollable geyser—of bottom-up source material. Subversion was already in the hands of thousands of early adopters, and those users were giving tons of feedback, not only about Subversion, but about its existing documentation.

During the entire time they wrote this book, Ben, Mike, and Brian haunted the Subversion mailing lists and chat rooms incessantly, carefully noting the problems users were having in real-life situations. Monitoring such feedback is part of their job descriptions at CollabNet anyway, and it gave them a huge advantage when they set out to document Subversion. The book they produced is grounded firmly in the bedrock of experience, not in the shifting sands of wishful thinking; it combines the best aspects of user manual and FAQ sheet. This duality might not be noticeable on a first reading. Taken in order, front to back, the book is simply a straightforward description of a piece of software. There's the overview, the obligatory guided tour, the chapter on administrative configuration, some advanced topics, and of course a command reference and troubleshooting guide. Only when you come back to it later, seeking the solution to some specific problem, does its authenticity shine out: the telling details that can only result from encounters with the unexpected, the examples honed from genuine use cases, and most of all the sensitivity to the user's needs and the user's point of view.

Of course, no one can promise that this book will answer every question you have about Subversion.

Sometimes, the precision with which it anticipates your questions will seem eerily telepathic; yet occasionally, you will stumble into a hole in the community's knowledge, and come away empty-handed. When this happens, the best thing you can do is email `<users@subversion.tigris.org>` and present your problem. The authors are still there, still watching, and they include not just the three listed on the cover, but many others who contributed corrections and original material. From the community's point of view, solving your problem is merely a pleasant side effect of a much larger project—namely, slowly adjusting this book, and ultimately Subversion itself, to more closely match the way people actually use it. They are eager to hear from you not merely because they can help you, but because you can help them. With Subversion as with all active free software projects, *you are not alone*.

Let this book be your first companion.

— Karl Fogel, Chicago, 14 March, 2004

Preface

> "If C gives you enough rope to hang yourself, think of Subversion as a sort of rope storage facility." —Brian W. Fitzpatrick

In the world of open-source software, the Concurrent Versions System (CVS) has long been the tool of choice for version control. And rightly so. CVS itself is free software, and its non-restrictive

Experienced System Administrators
The assumption here is that you've probably used CVS before, and are dying to get a Subversion server up and running ASAP. Chapter 5, *Repository Administration* and Chapter 6, *Server Configuration* will show you how to create your first repository and make it available over the network. After that's done, Chapter 3, *Guided Tour* and Appendix A, *Subversion for CVS Users* are the fastest routes to learning the Subversion client while drawing on your CVS experience.

New users
Your administrator has probably set up Subversion already, and you need to learn how to use the client. If you've never used a version control system (like CVS), then Chapter 2, *Basic Concepts* and Chapter 3, *Guided Tour* are a vital introduction. If you're already an old hand at CVS, chapter 3 and appendix A are the best place to start.

Advanced users
Whether you're a user or administrator, eventually your project will grow larger. You're going to want to learn how to do more advanced things with Subversion, such as how to use branches and perform merges (Chapter 4, *Branching and Merging*), how to use Subversion's property support, how to configure runtime options (Chapter 7, *Advanced Topics*), and other things. These two chapters aren't vital at first, but be sure to read them once you're comfortable with the basics.

Developers
Presumably, you're already familiar with Subversion, and now want to either extend it or build new software on top of its many APIs. Chapter 8, *Developer Information* is just for you.

The book ends with reference material—Chapter 9, *Subversion Complete Reference* is a reference guide for all Subversion commands, and the appendices cover a number of useful topics. These are the chapters you're mostly likely to come back to after you've finished the book.

Conventions Used in This Book

This section covers the various conventions used in this book.

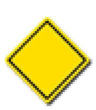
Typographic Conventions

Constant width
Used for commands, command output, and switches

`Constant width italic`
Used for replaceable items in code and text

`Italic`
Used for file and directory names

Icons



Note

This icon designates a note relating to the surrounding text.

Tip

This icon designates a helpful tip relating to the surrounding text.

Warning

This icon designates a warning relating to the surrounding text.

Note that 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, not necessarily serve as examples of good programming style.

Organization of This Book

The chapters that follow and their contents are listed here:

Chapter 1, *Introduction*
Covers the history of Subversion as well as its features, architecture, components, and install methods. Also includes a quick-start guide.

Chapter 2, *Basic Concepts*
Explains the basics of version control and different versioning models, along with Subversion's repository, working copies, and revisions.

Chapter 3, *Guided Tour*
Walks you through a day in the life of a Subversion user. It demonstrates how to use Subversion to obtain, modify, and commit data.

Chapter 4, *Branching and Merging*
Discusses branches, merges, and tagging, including best practices for branching and merging, common use cases, how to undo changes, and how to easily swing from one branch to the next.

Chapter 5, *Repository Administration*
Describes the basics of the Subversion repository, how to create, configure, and maintain a repository, and the tools you can use to do all of this.

Chapter 6, *Server Configuration*
Explains how to configure your Subversion server and the three ways to access your repository: `HTTP`, the `svn` protocol, and local access. It also covers the details of authentication, authorization and anonymous access.

Chapter 7, *Advanced Topics*
Explores the Subversion client configuration files, file and directory properties, how to `ignore` files in your working copy, how to include external trees in your working copy, and lastly, how to handle vendor branches.

Chapter 8, *Developer Information*
Describes the internals of Subversion, the Subversion filesystem, and the working copy administrative areas from a programmer's point of view. Demonstrates how to use the public APIs to write a program that uses Subversion, and most importantly, how to contribute to the development of Subversion.

Chapter 9, *Subversion Complete Reference*
Explains in great detail every subcommand of **svn**, **svnadmin**, and **svnlook** with plenty of examples for the whole family!

Appendix A, *Subversion for CVS Users*
Covers the similarities and differences between Subversion and CVS, with numerous suggestions on how to

break all the bad habits you picked up from years of using CVS. Included are descriptions of Subversion revision numbers, versioned directories, offline operations, **update** vs. **status**, branches, tags, metadata, conflict resolution, and authentication.

Appendix B, *WebDAV and Autoversioning*
Describes the details of WebDAV and DeltaV, and how you can configure your Subversion repository to be mounted read/write as a DAV share.

Appendix C, *Third Party Tools*
Discusses tools that support or use Subversion, including alternative client programs, repository browser tools, and so on.

This Book is Free

This book started out as bits of documentation written by Subversion project developers, which were then coalesced into a single work and rewritten. As such, it has always been under a free license. (See Appendix D, *Copyright*.) In fact, the book was written in the public eye, as a part of Subversion. This means two things:

- You will always find the latest version of this book in the book's own Subversion repository.
- You can distribute and make changes to this book however you wish—it's under a free license. Of course, rather than distribute your own private version of this book, we'd much rather you send feedback and patches to the Subversion developer community. See the section called “Contributing to Subversion” to learn about joining this community.

A relatively recent online version of this book can be found at http://svnbook.red-bean.com.

Acknowledgments

This book would not be possible (nor very useful) if Subversion did not exist. For that, the authors would like to thank Brian Behlendorf and CollabNet for the vision to fund such a risky and ambitious new Open Source project; Jim Blandy for the original Subversion name and design—we love you, Jim; Karl Fogel for being such a good friend and a great community leader, in that order.[1]

Thanks to O'Reilly and our editors, Linda Mui and Tatiana Diaz for their patience and support.

Finally, we thank the countless people who contributed to this book with informal reviews, suggestions, and fixes: While this is undoubtedly not a complete list, this book would be incomplete and incorrect without the help of: Jani Averbach, Ryan Barrett, Francois Beausoleil, Jennifer Bevan, Matt Blais, Zack Brown, Martin Buchholz, Brane Cibej, John R. Daily, Peter Davis, Olivier Davy, Robert P. J. Day, Mo DeJong, Brian Denny, Joe Drew, Nick Duffek, Ben Elliston, Justin Erenkrantz, Shlomi Fish, Julian Foad, Chris Foote, Martin Furter, Dave Gilbert, Eric Gillespie, Matthew Gregan, Art Haas, Greg Hudson, Alexis Huxley, Jens B. Jorgensen, Tez Kamihira, David Kimdon, Mark Benedetto King, Andreas J. Koenig, Nuutti Kotivuori, Matt Kraai, Scott Lamb, Vincent Lefevre, Morten Ludvigsen, Paul Lussier, Bruce A. Mah, Philip Martin, Feliciano Matias, Patrick Mayweg, Gareth McCaughan, Jon Middleton, Tim Moloney, Mats Nilsson, Joe Orton, Amy Lyn Pilato, Kevin Pilch-Bisson, Dmitriy Popkov, Michael Price, Mark Proctor, Steffen Prohaska, Daniel Rall, Tobias Ringstrom, Garrett Rooney, Joel Rosdahl, Christian Sauer, Larry Shatzer, Russell Steicke, Sander Striker, Erik Sjoelund, Johan Sundstroem, John Szakmeister, Mason Thomas, Eric Wadsworth, Colin Watson, Alex Waugh, Chad Whitacre, Josef Wolf, Blair Zajac, and the entire Subversion community.

[1]Oh, and thanks, Karl, for being too overworked to write this book yourself.

From Ben Collins-Sussman

Chapter 1. Introduction

Version control is the art of managing changes to information. It has long been a critical tool for programmers, who typically spend their time making small changes to software and then undoing those changes the next day. But the usefulness of version control software extends far beyond the bounds of the software development world. Anywhere you can find people using computers to manage information that changes often, there is room for version control. And that's where Subversion comes into play.

This chapter contains a high-level introduction to Subversion—what it is; what it does; how to get it.

What is Subversion?

Subversion is a free/open-source version control system. That is, Subversion manages files and directories over time. A tree of files is placed into a central

[1]There's also a CollabNet Team Edition (CTE) offering aimed at smaller groups.

Behlendorf and Jason Robbins of CollabNet, and Greg Stein (at the time an independent developer active in the WebDAV/DeltaV specification process), Subversion quickly attracted a community of active developers. It turned out that many people had had the same frustrating experiences with CVS, and welcomed the chance to finally do something about it.

The original design team settled on some simple goals. They didn't want to break new ground in version control methodology, they just wanted to fix CVS. They decided that Subversion would match CVS's features, and preserve the same development model, but not duplicate CVS's most obvious flaws. And although it did not need to be a drop-in replacement for CVS, it should be similar enough that any CVS user could make the switch with little effort.

After fourteen months of coding, Subversion became “self-hosting” on August 31, 2001. That is, Subversion developers stopped using CVS to manage Subversion's own source code, and started using Subversion instead.

While CollabNet started the project, and still funds a large chunk of the work (it pays the salaries of a few full-time Subversion developers), Subversion is run like most open-source projects, governed by a loose, transparent set of rules that encourage meritocracy. CollabNet's copyright license is fully compliant with the Debian Free Software Guidelines. In other words, anyone is free to download, modify, and redistribute Subversion as he pleases; no permission from CollabNet or anyone else is required.

Subversion's Features

When discussing the features that Subversion brings to the version control table, it is often helpful to speak of them in terms of how they improve upon CVS's design. If you're not familiar with CVS, you may not understand all of these features. And if you're not familiar with version control at all, your eyes may glaze over unless you first read Chapter 2, *Basic Concepts*, in which we provide a gentle introduction to version control in general.

Subversion provides:

Directory versioning

CVS only tracks the history of individual files, but Subversion implements a “virtual” versioned filesystem that tracks changes to whole directory trees over time. Files *and* directories are versioned.

True version history

Since CVS is limited to file versioning, operations such as copies and renames—which might happen to files, but which are really changes to the contents of some containing directory—aren't supported in CVS. Additionally, in CVS you cannot replace a versioned file with some new thing of the same name without the new item inheriting the history of the old—perhaps completely unrelated—file. With Subversion, you can add, delete, copy, and rename both files and directories. And every newly added file begins with a fresh, clean history all its own.

Atomic commits

A collection of modifications either goes into the repository completely, or not at all. This allows developers to construct and commit changes as logical chunks, and prevents problems that can occur when only a portion of a set of changes is successfully sent to the repository.

Versioned metadata

Each file and directory has a set of properties—keys and their values—associated with it. You can create and store any arbitrary key/value pairs you wish. Properties are versioned over time, just like file contents.

Choice of network layers

Subversion has an abstracted notion of repository access, making it easy for people to implement new network mechanisms. Subversion can plug into the Apache HTTP Server as an extension module. This gives Subversion a big advantage in stability and interoperability, and instant access to existing features provided by that server—authentication, authorization, wire compression, and so on. A more lightweight, standalone Subversion server process is also available. This server speaks a custom protocol which can be easily tunneled over SSH.

Consistent data handling
Subversion expresses file differences using a binary differencing algorithm, which works identically on both text (human-readable) and binary (human-unreadable) files. Both types of files are stored equally compressed in the repository, and differences are transmitted in both directions across the network.

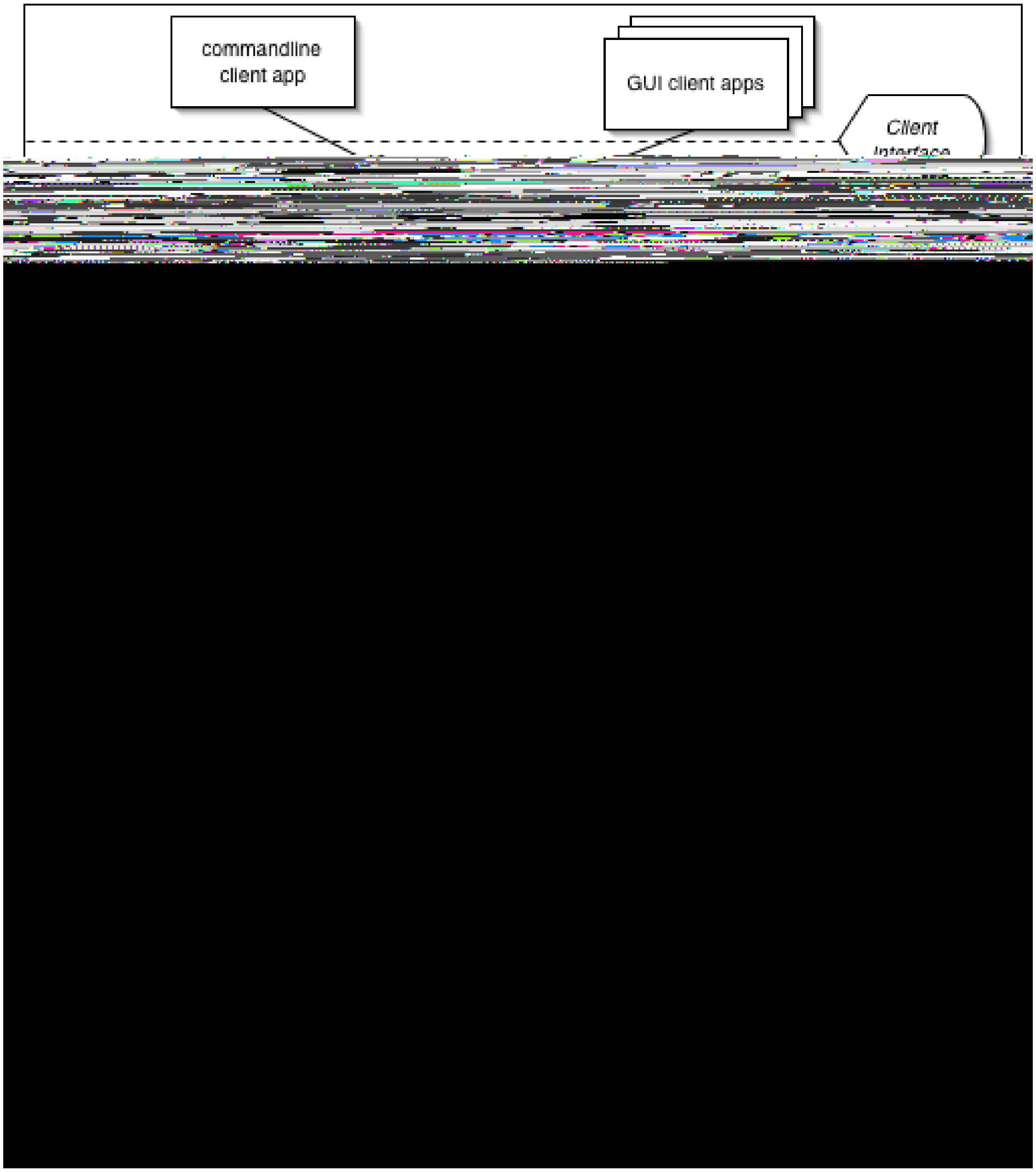
Efficient branching and tagging
The cost of branching and tagging need not be proportional to the project size. Subversion creates branches and tags by simply copying the project, using a mechanism similar to a hard-link. Thus these operations take only a very small, constant amount of time.

Hackability
Subversion has no historical baggage; it is implemented as a collection of shared C libraries with well-defined APIs. This makes Subversion extremely maintainable and usable by other applications and languages.

Subversion's Architecture

Figure 1.1, “Subversion's Architecture” illustrates what one might call a “mile-high” view of Subversion's design.

Figure 1.1. Subversion's Architecture



On one end is a Subversion repository that holds all of your versioned data. On the other end is your Subversion client program, which manages local reflections of portions of that versioned data (called “working copies”). Between these extremes are multiple routes through various Repository Access (RA) layers. Some of these routes go across computer networks and through network servers which then access the repository. Others bypass the network altogether and access the repository directly.

Installing Subversion

Subversion is built on a portability layer called APR—the Apache Portable Runtime library. The APR library provides all the interfaces that Subversion needs to function on different operating systems: disk access, network access, memory management, and so on. While Subversion is able to use Apache as one of its network server programs, its dependence on APR *does not* mean that Apache is a required component. APR is a standalone library useable by any application. It does mean, however, that like Apache, Subversion clients and servers run on any operating system that the Apache httpd server runs on: Windows, Linux, all flavors of BSD, Mac OS X, Netware, and others.

The easiest way to get Subversion is to download a binary package built for your operating system. Subversion's website (http://subversion.tigris.org) often has these packages available for download, posted by volunteers. The site usually contains graphical installer packages for users of Microsoft operating systems. If you run a Unix-like operating system, you can use your system's native package distribution system (RPMs, DEBs, the ports tree, etc.) to get Subversion.

Alternately, you can build Subversion directly from source code. From the Subversion website, download the latest source-code release. After unpacking it, follow the instructions in the `INSTALL` file to build it. Note that a released source package contains everything you need to build a command-line client capable of talking to a remote repository (in particular, the apr, apr-util, and neon libraries). But optional portions of Subversion have many other dependencies, such as Berkeley DB and possibly Apache httpd. If you want to do a complete build, make sure you have all of the packages documented in the `INSTALL` file. If you plan to work on Subversion itself, you can use your client program to grab the latest, bleeding-edge source code. This is documented in the section called "Get the Source Code".

Subversion's Components

Subversion, once installed, has a number of different pieces. The following is a quick overview of what you get. Don't be alarmed if the brief descriptions leave you scratching your head—there are *plenty* more pages in this book devoted to alleviating that confusion.

svn
: The command-line client program.

svnversion
: A program for reporting the state (in terms of revisions of the items present) of a working copy.

svnlook
: A tool for inspecting a Subversion repository.

svnadmin
: A tool for creating, tweaking or repairing a Subversion repository.

svndumpfilter
: A program for filtering Subversion repository dump streams.

mod_dav_svn
: A plug-in module for the Apache HTTP Server, used to make your repository available to others over a network.

svnserve
: A custom standalone server program, runnable as a daemon process or invokable by SSH; another way to make your repository available to others over a network.

stration will get you up and running. Along the way, we give links to the relevant chapters of this book.

If you're new to the entire concept of version control or to the “copy-modify-merge” model used by both CVS and Subversion, then you should read Chapter 2, *Basic Concepts* before going any further.

Note

The following example assumes that you have **svn**, the Subversion command-line client, and **svnadmin**, the administrative tool, ready to go. It also assumes you are using Subversion 1.2 or later (run **svn --version** to check.)

Subversion stores all versioned data in a central repository. To begin, create a new repository:

```
$ svnadmin create /path/to/repos
$ ls /path/to/repos
conf/  dav/  db/  format  hooks/  locks/  README.txt
```

This command creates a new directory `/path/to/repos` which contains a Subversion repository. This new directory contains (among other things) a collection of database files. You won't see your versioned files if you peek inside. For more information about repository creation and maintenance, see Chapter 5, *Repository Administration*.

Subversion has no concept of a “project”. The repository is just a virtual versioned filesystem, a large tree that can hold anything you wish. Some administrators prefer to store only one project in a repository, and others prefer to store multiple projects in a repository by placing them into separate directories. The merits of each approach are discussed in the section called “Choosing a Repository Layout”. Either way, the repository only manages files and directories, so it's up to humans to interpret particular directories as “projects”. So while you might see references to projects throughout this book, keep in mind that we're only ever talking about some directory (or collection of directories) in the repository.

In this example, we assume that you already have some sort of project (a collection of files and directories) that you wish to import into your newly created Subversion repository. Begin by organizing them into a single directory called `myproject` (or whatever you wish). For reasons that will be clear later (see Chapter 4, *Branching and Merging*), your project's tree structure should contain three top-level directories named `branches`, `tags`, and `trunk`. The `trunk` directory should contain all of your data, while `branches` and `tags` directories are empty:

```
/tmp/myproject/branches/
/tmp/myproject/tags/
/tmp/myproject/trunk/
                     foo.c
                     bar.c
                     Makefile
                     …
```

The `branches`, `tags`, and `trunk` subdirectories aren't actually required by Subversion. They're merely a popular convention that you'll most likely want to use later on.

Once you have your tree of data ready to go, import it into the repository with the **svn import** command (see the section called “**svn import**”):

```
$ svn import /tmp/myproject file:///path/to/repos/myproject -m "initial import"
Adding         /tmp/myproject/branches
Adding         /tmp/myproject/tags
```

```
Adding         /tmp/myproject/trunk
Adding         /tmp/myproject/trunk/foo.c
Adding         /tmp/myproject/trunk/bar.c
Adding         /tmp/myproject/trunk/Makefile
…
Committed revision 1.
$
```

Now the repository contains this tree of data. As mentioned earlier, you won't see your files by directly peeking into the repository; they're all stored within a database. But the repository's imaginary filesystem now contains a top-level directory named `myproject`, which in turn contains your data.

Note that the original `/tmp/myproject` directory is unchanged; Subversion is unaware of it. (In fact, you can even delete that directory if you wish.) In order to start manipulating repository data, you need to create a new "working copy" of the data, a sort of private workspace. Ask Subversion to "check out" a working copy of the `myproject/trunk` directory in the repository:

```
$ svn checkout file:///path/to/repos/myproject/trunk myproject
A  myproject/foo.c
A  myproject/bar.c
A  myproject/Makefile
…
Checked out revision 1.
```

Now you have a personal copy of part of the repository in a new directory named `myproject`. You can edit the files in your working copy and then commit those changes back into the repository.

- Enter your working copy and edit a file's contents.
- Run **svn diff** to see unified diff output of your changes.
- Run **svn commit** to commit the new version of your file to the repository.
- Run **svn update** to bring your working copy "up-to-date" with the repository.

For a full tour of all the things you can do with your working copy, read Chapter 3, *Guided Tour*.

At this point, you have the option of making your repository available to others over a network. See Chapter 6, *Server Configuration* to learn about the different sorts of server processes available and how to configure them.

Chapter 2. Basic Concepts

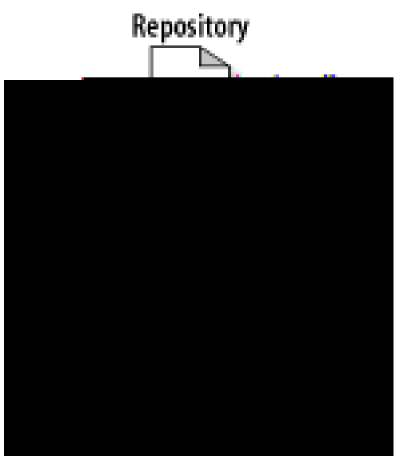
This chapter is a short, casual introduction to Subversion. If you're new to version control, this chapter is definitely for you. We begin with a discussion of general version control concepts, work our way into the specific ideas behind Subversion, and show some simple examples of Subversion in use.

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” illustrates this.

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 he 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

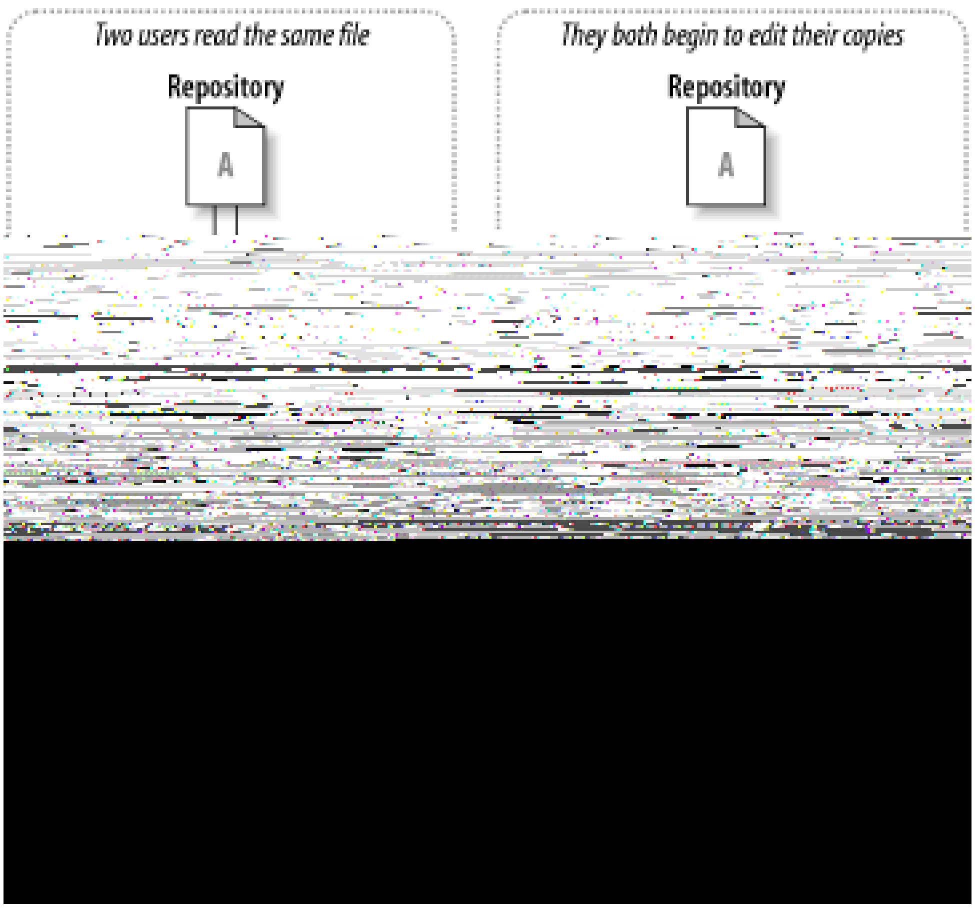
The core mission of a version control system is to enable collaborative editing and sharing of data. But different systems use different strategies to achieve this.

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 the scenario shown in Figure 2.2, “The problem to avoid”. Suppose we have two co-workers, Harry and Sally. They each decide to edit the same repository file at the same time. If Harry saves his changes to the repository first, then it's possible that (a few moments later) Sally could accidentally overwrite them with her own new version of the file. While Harry's version of the file won't be lost forever (because the system remembers every change), any changes Harry made *won't* be present in Sally's newer version of the file, because she never saw Harry's changes to begin with. Harry'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!

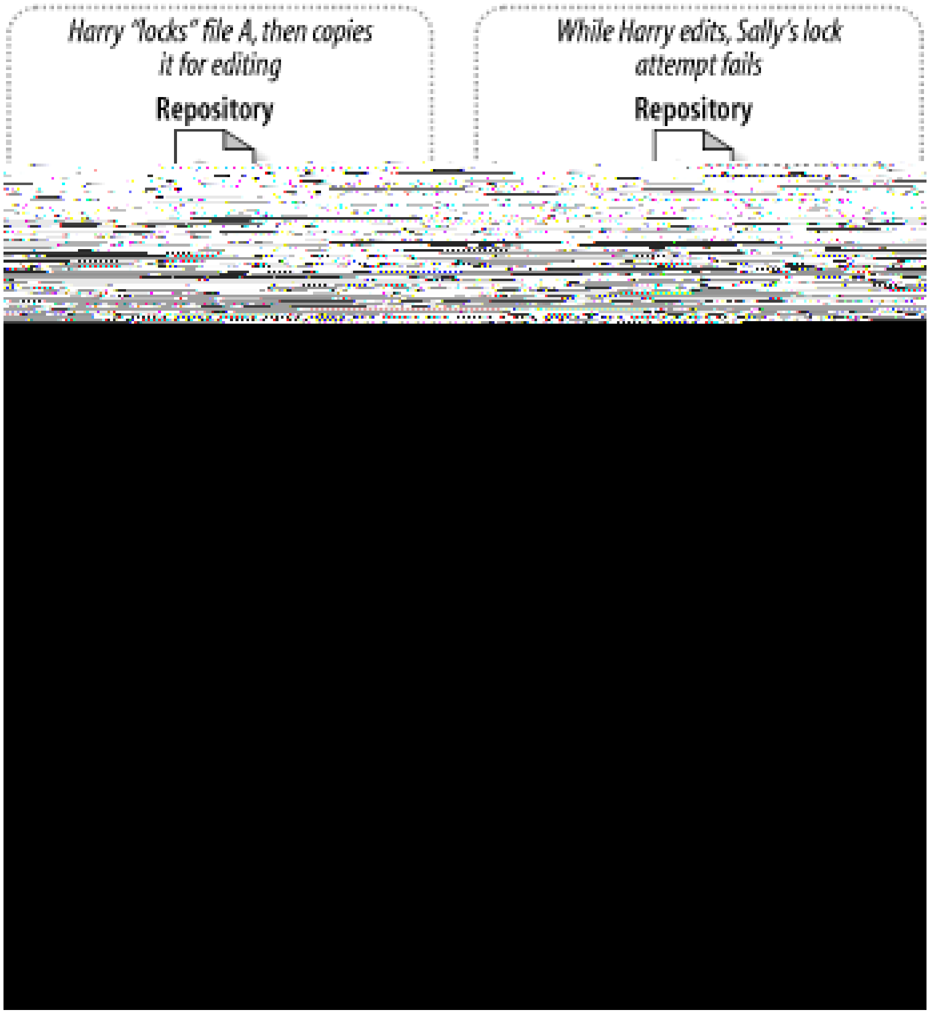
Figure 2.2. The problem to avoid



The Lock-Modify-Unlock Solution

Many version control systems use a *lock-modify-unlock* model to address the problem of many authors clobbering each other's work. In this model, the repository allows only one person to change a file at a time. This exclusivity policy is managed using locks. Harry must “lock” a file before he can begin making changes to it. If Harry has locked a file, then Sally cannot also lock it, and therefore cannot make any changes to that file. All she can do is read the file, and wait for Harry to finish his changes and release his lock. After Harry unlocks the file, Sally can take her turn by locking and editing the file. Figure 2.3, “The lock-modify-unlock solution” demonstrates this simple solution.

Figure 2.3. The lock-modify-unlock solution



more. The locking system was powerless to prevent the problem—yet it somehow provided a false sense of security. It's easy for Harry and Sally to imagine that by locking files, each is beginning a safe, insulated task, and thus not bother discussing their incompatible changes early on.

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 contacts the project repository and creates a personal *working copy*—a local reflection of the repository's files and directories. 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 Harry and Sally 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. Sally saves her changes to the repository first. When Harry 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 Harry asks his client to *merge* any new changes from the repository into his working copy of file A. Chances are that Sally'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” and Figure 2.5, “The copy-modify-merge solution (continued)” show this process.

Figure 2.4. The copy-modify-merge solution

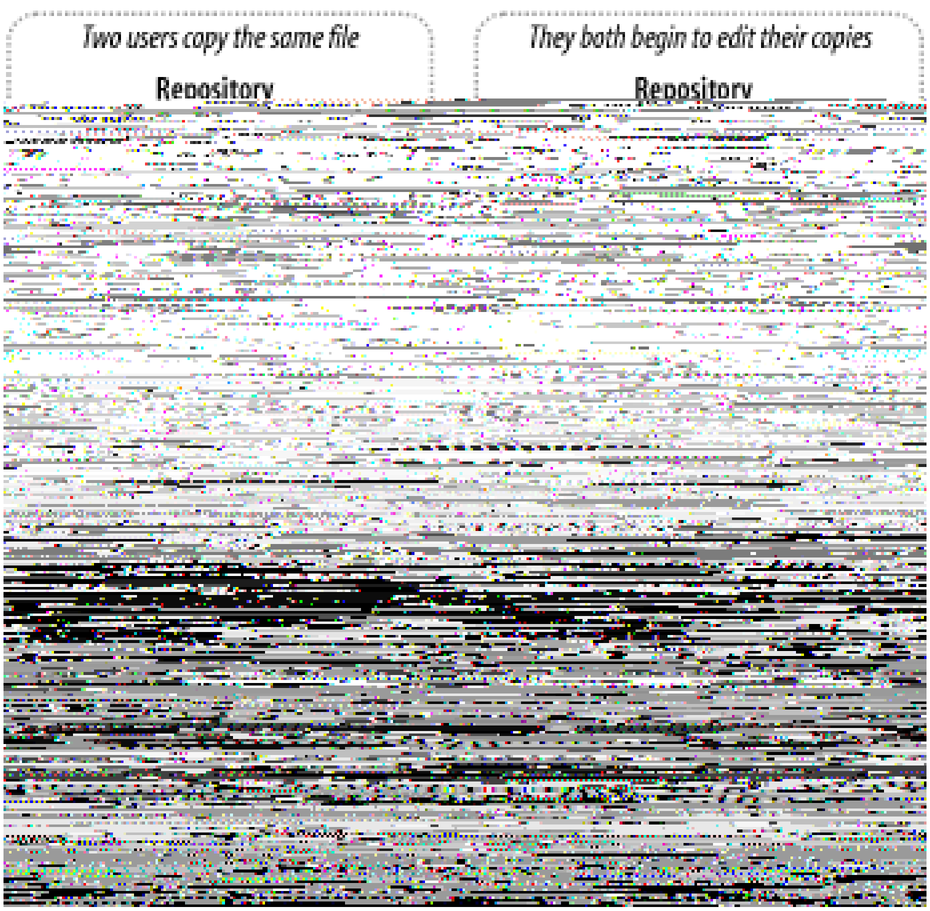
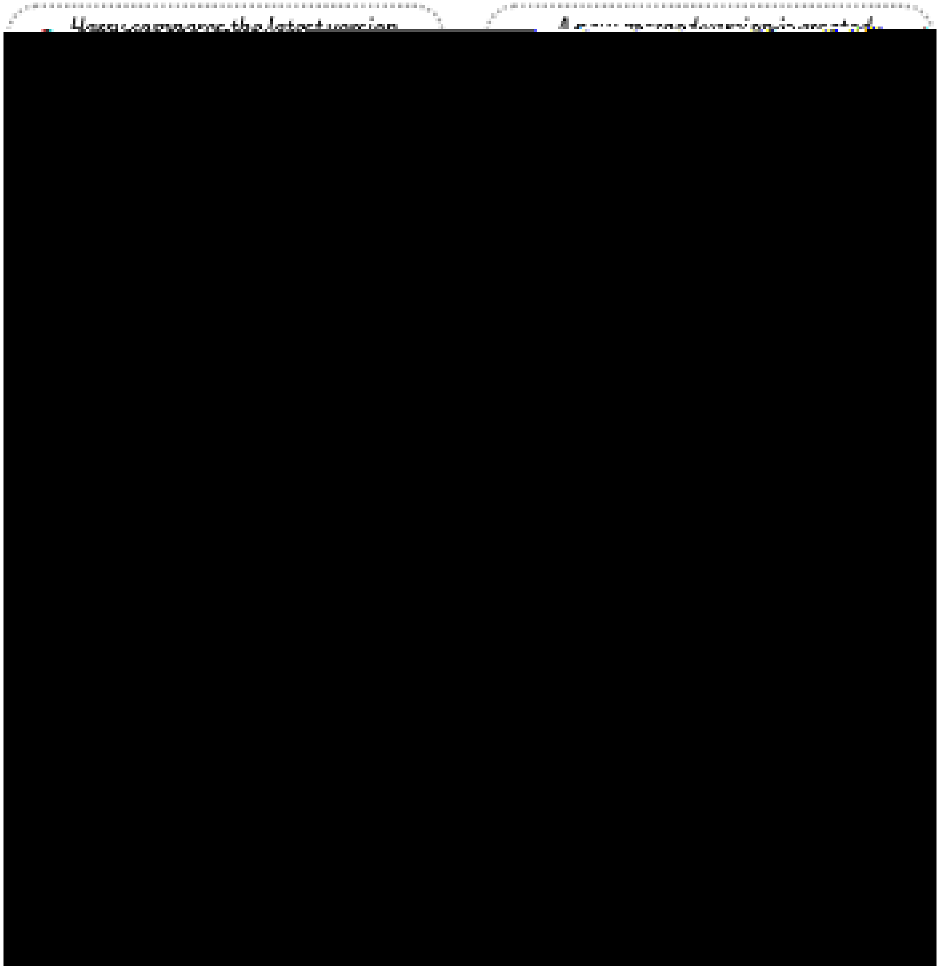


Figure 2.5. The copy-modify-merge solution (continued)



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

When Locking is Necessary

While the lock-modify-unlock model is considered generally harmful to collaboration, there are still times when locking is appropriate.

The copy-modify-merge model is based on the assumption that files are contextually mergeable: that is, that the majority of the files in the repository are line-based text files (such as program source code). But for files with binary formats, such as artwork or sound, it's often impossible to merge conflicting changes. In these situations, it really is necessary to users to take strict turns when changing the file. Without serialized access, somebody ends up wasting time on changes that are ultimately discarded.

While CVS and Subversion are still primarily copy-modify-merge systems, they both recognize the need to lock an occasional file and provide mechanisms for this. See the section called “Locking”.

Subversion in Action

It's time to move from the abstract to the concrete. In this section, we'll show real examples of Subversion being used.

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. You can even have multiple working copies of the same project.

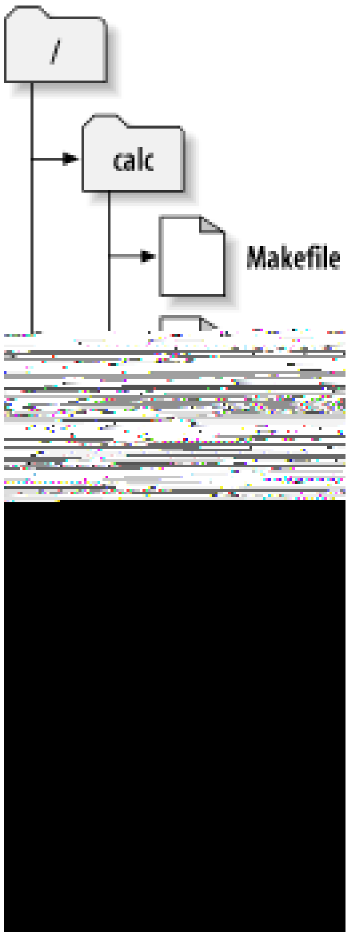
After you've made some changes to the files in your working copy and verified that they work properly, Subversion provides you with commands to “publish” your changes to the other people working with you on your project (by writing to the repository). If other people publish their own changes, Subversion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

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

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

For example, suppose you have a repository that contains two software projects, `paint` and `calc`. Each project lives in its own top-level subdirectory, as shown in Figure 2.6, “The repository's filesystem”.

Figure 2.6. The repository's filesystem



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

```
$ svn checkout http://svn.example.com/repos/calc
A    calc/Makefile
A    calc/integer.c
A    calc/button.c
Checked out revision 56.

$ ls -A calc
Makefile  integer.c  button.c  .svn/
```

The list of letter A's indicates that Subversion is adding a number of items to your working copy. You now have a personal copy of the repository's `/calc` directory, with one additional entry—`.svn`—which holds the extra information needed by Subversion, as mentioned earlier.

Repository URLs

Subversion repositories can be accessed through many different methods—on local disk, or through various network protocols. A repository location, however, is always a URL. Table 2.1, “Repository Access URLs” describes how different URL schemas map to the available access methods.

Table 2.1. Repository Access URLs

Schema	Access Method
`file:///`	direct repository access (on local disk)
`http://`	access via WebDAV protocol to Subversion-aware Apache server
`https://`	same as `http://`, but with SSL encryption.
`svn://`	access via custom protocol to an `svnserve` server
`svn+ssh://`	same as `svn://`, but through an SSH tunnel.

For more information on how Subversion parses URLs, see the section called “Subversion Repository URLs”.

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

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

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

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

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

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

```
$ pwd
/home/sally/calc

$ ls -A
.svn/ Makefile integer.c button.c

$ svn update
U    button.c
Updated to revision 57.
```

The output from the **svn update** command indicates that Subversion updated the contents of `button.c`. Note that Sally 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

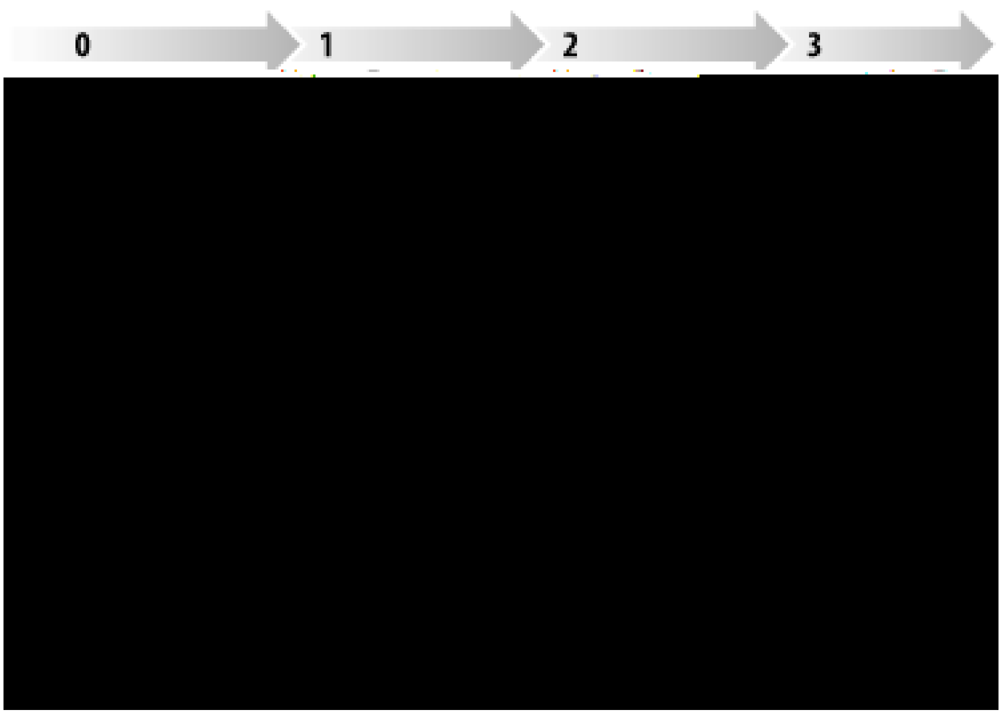
An **svn commit** operation can publish changes to any number of files and directories as a single atomic transaction. In your working copy, you can change files' contents, create, delete, rename and copy files and directories, and then commit the complete set of changes as a unit.

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.

Figure 2.7, "The repository" illustrates a nice way to visualize the repository. Imagine an array of revision numbers, starting at 0, stretching from left to right. Each revision number has a filesystem tree hanging below it, and each tree is a "snapshot" of the way the repository looked after a 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

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:

```
calc/Makefile:4
     integer.c:4
     button.c:4
```

At the moment, this working directory corresponds exactly to revision 4 in the repository. However, suppose you make a change to `button.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:

```
calc/Makefile:4
     integer.c:4
     button.c:5
```

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

```
calc/Makefile:6
     integer.c:6
     button.c:6
```

Sally's change to `integer.c` will appear in your working copy, and your change will still be present in `button.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. An **svn commit** of the file will do nothing, and an **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 an **svn commit** of the file will succeed in publishing your changes, and an **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. An **svn commit** of the file will do nothing, and an **svn update** of the file will 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. An **svn commit** of the file will fail with an “out-of-date” error. The file should be updated first; an **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 the **svn status** command will show you the state of any item in your working copy. For more information on that command, see the section called “**svn status**”.

Mixed Revision Working Copies

As a general principle, Subversion tries to be as flexible as possible. One special kind of flexibility is the ability to have a working copy containing files and directories with a mix of different working revision numbers. Unfortunately, this flexibility tends to confuse a number of new users. If the earlier example showing mixed revisions perplexed you, here's a primer on both why the feature exists and how to make use of it.

Updates and Commits are Separate

One of the fundamental rules of Subversion is that a “push” action does not cause a “pull”, nor the other way around. Just because you're ready to submit new changes to the repository doesn't mean you're ready to receive changes from other people. And if you have new changes still in progress, then **svn update** should gracefully merge repository changes into your own, rather than forcing you to publish them.

The main side-effect of this rule is that it means a working copy has to do extra bookkeeping to track mixed revisions, and be tolerant of the mixture as well. It's made more complicated by the fact that directories themselves are versioned.

For example, suppose you have a working copy entirely at revision 10. You edit the file `foo.html` and then perform an **svn commit**, which creates revision 15 in the repository. After the commit succeeds, many new users would expect the working copy to be entirely at revision 15, but that's not the case! Any number of changes might have happened in the repository between revisions 10 and 15. The client knows nothing of those changes in the repository, since you haven't yet run **svn update**, and **svn commit** doesn't pull down new changes. If, on the other hand, **svn commit** *were* to automatically download the newest changes, then it would be possible to set the entire working copy to revision 15—but then we'd be breaking the fundamental rule of “push” and “pull” remaining separate actions. Therefore the only safe thing the Subversion client can do is mark the one file—`foo.html`—as being at revision 15. The rest of the working copy remains at revision 10. Only by running **svn update** can the latest changes be downloaded, and the whole working copy be marked as revision 15.

Mixed revisions are normal

The fact is, *every time* you run **svn commit**, your working copy ends up with some mixture of revisions. The things you just committed are marked as having larger working revisions than everything else. After several commits (with no updates in-between) your working copy will contain a whole mixture of revisions. Even if you're the only person using the repository, you will still see this phenomenon. To examine your mixture of working revisions, use the **svn status --verbose** command (see the section called “**svn status**” for more information.)

Often, new users are completely unaware that their working copy contains mixed revisions. This can be confusing, because many client commands are sensitive to the working revision of the item they're examining. For example, the **svn log** command is used to display the history of changes to a file or directory (see the section called “**svn log**”). When the user invokes this command on a working copy object, they expect to see the entire history of the object. But if the object's working revision is quite old (often because **svn update** hasn't been run in a long time), then the history of the *older* version of the object is shown.

Mixed revisions are useful

If your project is sufficiently complex, you'll discover that it's sometimes nice to forcibly “backdate” portions of your working copy to an earlier revision; you'll learn how to do that in Chapter 3. Perhaps you'd like to test an earlier version of a sub-module contained in a subdirectory, or perhaps you'd like to figure out when a bug first came into existence in a specific file. This is the “time machine” aspect of a version control system — the feature which allows you to move any portion of your working copy forward and backward in history.

Mixed revisions have limitations

However you make use of mixed revisions in your working copy, there are limitations to this flexibility.

First, you cannot commit the deletion of a file or directory which isn't fully up-to-date. If a newer version of the item exists in the repository, your attempt to delete will be rejected, to prevent you from accidentally destroying changes you've not yet seen.

Second, you cannot commit a metadata change to a directory unless it's fully up-to-date. You'll learn about attaching “properties” to items in Chapter 6. A directory's working revision defines a specific set of entries and properties, and thus committing a property change to an out-of-date directory may destroy properties you've not yet seen.

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 detailed tour of Subversion's commands and features.

Chapter 3. Guided Tour

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

Note that this chapter is not meant to be an exhaustive list of all Subversion's commands—rather, it's a conversational introduction to the most common Subversion tasks you'll encounter. This chapter assumes that you've read and understood Chapter 2, *Basic Concepts* and are familiar with the general model of Subversion. For a complete reference of all commands, see Chapter 9, *Subversion Complete Reference*.

Help!

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

Import

You use **svn import** to import a new project into a Subversion repository. While this is most likely the very first thing you will do when you set up your Subversion server, it's not something that happens very often. For a detailed description of import, see the section called "**svn import**" later in this chapter.

Revisions: Numbers, Keywords, and Dates, Oh My!

Before we go on, you should know a bit about how to identify a particular revision in your repository. As you learned in the section called "Revisions", a revision is a "snapshot" of the repository at a particular moment in time. As you continue to commit and grow your repository, you need a mechanism for identifying these snapshots.

You specify these revisions by using the `--revision` (`-r`) switch plus the revision you want (**svn --revision REV**) or you can specify a range by separating two revisions with a colon (

do that later in this chapter), you can refer to it as “3”.

Revision Keywords

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

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

These keywords allow you to perform many common (and helpful) operations without having to look up specific revision numbers or remember the exact revision of your working copy.

Revision Dates

Anywhere that you specify a revision number or revision keyword, you can also specify a date inside

```
$ svn log --revision {2002-11-20}:{2002-11-29}
…
```

As we pointed out, you can also mix dates and revisions:

```
$ svn log --revision {2002-11-20}:4040
```

Users should be aware of a subtlety that can become quite a stumbling-block when dealing with dates in Subversion. Since the timestamp of a revision is stored as a property of the revision—an unversioned, modifiable property—revision timestamps can be changed to represent complete falsifications of true chronology, or even removed altogether. This will wreak havoc on the internal date-to-revision conversion that Subversion performs.

Initial Checkout

Most of the time, you will start using a Subversion repository by doing a *checkout* of your project. Checking out a repository creates a copy of it on your local machine. This copy contains the `HEAD` (latest revision) of the Subversion repository that you specify on the command line:

```
$ svn checkout http://svn.collab.net/repos/svn/trunk
A  trunk/subversion.dsw
```

the repository. While these aren't strict requirements like the syntax described above, they help to organize frequently performed tasks. The `/trunk` part of URLs you'll find throughout this book is one of these conventions; we'll talk a lot more about it and related recommendations in Chapter 4, *Branching and Merging*.

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

```
$ svn checkout http://svn.collab.net/repos/svn/trunk/doc/book/tools
A  tools/readme-dblite.html
A  tools/fo-stylesheet.xsl
A  tools/svnbook.el
A  tools/dtd
A  tools/dtd/dblite.dtd
…
Checked out revision 2499.
```

Since Subversion uses a “copy-modify-merge” model instead of “lock-modify-unlock” (see Chapter 2, *Basic Concepts*), you're already able to start making changes to the files and directories in your working copy. Your working copy is just like any other collection of files and directories on your system. You can edit and change them, move them around, you can even delete the entire working copy and forget about it.

Note

While your working copy is “just like any other collection of files and directories on your system”, you need to let Subversion know if you're going to be rearranging anything inside of your working copy. If you want to copy or move an item in a working copy, you should use **svn copy** or **svn move** instead of the copy and move commands provided by your operating system. We'll talk more about them later in this chapter.

Unless you're ready to commit a new file or directory, or changes to existing ones, there's no need to further notify the Subversion server that you've done anything.

What's with the `.svn` directory?

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

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

```
$ svn checkout http://svn.collab.net/repos/svn/trunk subv
A  subv/subversion.dsw
A  subv/svn_check.dsp
A  subv/COMMITTERS
A  subv/configure.in
A  subv/IDEAS
…
Checked out revision 2499.
```

That will place your working copy in a directory named `subv` instead of a directory named `trunk` as we did previously.

Basic Work Cycle

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
  - **svn update**
- Make changes
  - **svn add**
  - **svn delete**
  - **svn copy**
  - **svn move**
- Examine your changes
  - **svn status**
  - **svn diff**
  - **svn revert**
- Merge others' changes into your working copy
  - **svn update**
  - **svn resolved**
- Commit your changes
  - **svn commit**

Update Your Working Copy

When working on a project with a team, you'll want to update your working copy to receive any changes made since your last update by other developers on the project. Use **svn update** to bring your working copy into sync with the latest revision in the repository.

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

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

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

`U foo`
File `foo` was 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 repository considers them to be distinct objects with distinct histories.

`G foo`
File `foo` received new changes from the repository, but your local copy of the file had your modifications. Either the changes did not intersect, or the changes were exactly the same as your local modifications, so Subversion has successfully 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 decide on a particular change (or set of changes) to make, such as writing a new feature, fixing a bug, etc. The Subversion commands that you will use here are **svn add**, **svn delete**, **svn copy**, and **svn move**. However, if you are merely editing files that are already in Subversion, you may not need to use any of these commands until you commit. Changes you can make to your working copy:

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

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

To make file changes, use your text editor, word processor, graphics program, or whatever tool you would normally use. Subversion handles binary files just as easily as it handles text files—and just as efficiently too.

1

tionally, this is a good opportunity to review and scrutinize changes before publishing them. You can see exactly what changes you've made by using **svn status**, **svn diff**, and **svn revert**. You will usually use the first two commands to find out what files have changed in your working copy, and then perhaps the third to revert some (or all) of those changes.

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

svn status

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

CVS Users: Hold That Update!

You're probably used to using **cvs update** to see what changes you've made to your working copy. **svn status** will give you all the information you need regarding what has changed in your working copy—without accessing the repository or potentially incorporating new changes published by other users.

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

If you run **svn status** at the top of your working copy with no arguments, it will detect all file and tree changes you've made. Below are examples of the different status codes that **svn status** can return. (Note that the text following # is not actually printed by **svn status**.)

```
  L     some_dir            # svn left a lock in the .svn area of some_dir
M       bar.c               # the content in bar.c has local modifications
 M      baz.c               # baz.c has property but no content modifications
X       3rd_party           # dir is part of an externals definition
?       foo.o               # svn doesn't manage foo.o
!       some_dir            # svn manages this, but it's missing or incomplete
~       qux                 # versioned as file/dir/link, but type has changed
I       .screenrc           # svn doesn't manage this, and is set to ignore it
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        # file is scheduled for deletion
A       stuff/loot/bloo.h   # file is scheduled for addition
C       stuff/loot/lump.c   # file has textual conflicts from an update
 C      stuff/loot/glub.c   # file has property conflicts from an update
R       xyz.c               # file is scheduled for replacement
    S   stuff/squawk        # file or dir has been switched to a branch
     K  dog.jpg             # file is locked locally; lock-token present
     O  cat.jpg             # file is locked in the repository by other user
     B  bird.jpg            # file is locked locally, but lock has been broken
     T  fish.jpg            # file is locked locally, but lock has been stolen
```

In this output format **svn status** prints five 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:

`A item`
The file, directory, or symbolic link `item` has been scheduled for addition into the repository.

`C item`
The file `item` is in a state of conflict. That is, changes received from the server during an update overlap with local changes that you have in your working copy. You must resolve this conflict before committing your changes to the repository.

`D item`
The file, directory, or symbolic link `item` has been scheduled for deletion from the repository.

`M item`
The contents of the file `item` have been modified.

`R item`
The file, directory, or symbolic link `item` has been scheduled to replace `item` in the repository. This means that the object is first deleted, then another object of the same name is added, all within a single revision.

`X item`
The directory `item` is unversioned, but is related to a Subversion externals definition. To find out more about externals definitions, see the section called "Externals Definitions".

`? item`
The file, directory, or symbolic link `item` is not under version control. You can silence the question marks by either passing the `--quiet` (`-q`) switch to **svn status**, or by setting the `svn:ignore` property on the parent directory. For more information on ignored files, see the section called "`svn:ignore`".

`! item`
The file, directory, or symbolic link `item` is under version control but is missing or somehow incomplete. The item can be missing if it's removed using a non-Subversion command. In the case of a directory, it can be incomplete if you happened to interrupt a checkout or update. A quick **svn update** will refetch the file or directory from the repository, or **svn revert file** will restore a missing file.

`~ item`
The file, directory, or symbolic link `item` is in the repository as one kind of object, but what's actually in your working copy is some other kind. For example, Subversion might have a file in the repository, but you removed the file and created a directory in its place, without using the **svn delete** or **svn add** command.

`I item`
The file, directory, or symbolic link `item` is not under version control, and Subversion is configured to ignore it during **svn add**, **svn import** and **svn status** operations. For more information on ignored files, see the section called "`svn:ignore`". Note that this symbol only shows up if you pass the `--no-ignore` option to **svn status**—otherwise the file would be ignored and not listed at all!

The second column tells the status of a file or directory's properties (see the section called "Properties" for more information on properties). If an `M` appears in the second column, then the properties have been modified, otherwise a whitespace will be printed.

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

The fourth column will only show whitespace or a `+` which means that the file or directory is scheduled to be added or modified with additional attached history. This typically happens when you **svn move** or **svn copy** a file or directory. If you see `A    +`, this means the item is scheduled for addition-with-history.

It could be a file, or the root of a copied directory. `+` means the item is part of a subtree scheduled for addition-with-history, i.e. some parent got copied, and it's just coming along for the ride. `M    +` means the item is part of a subtree scheduled for addition-with-history, *and* it has local modifications. When you commit, first the parent will be added-with-history (copied), which means this file will automatically exist in the copy. Then the local modifications will be uploaded into the copy.

The fifth column will only show whitespace or an `S`. This signifies that the file or directory has been switched from the path of the rest of the working copy (using **svn switch**) to a branch.

The sixth column shows information about locks, which is further explained in the section called "Locking". (These are not the same locks as the ones indicated by an `L` in the third column; see Three meanings of "lock".)

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

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

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

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

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

None of the above invocations to **svn status** contact the repository, they work only locally by comparing the metadata in the `.svn` directory with the working copy. Finally, there is the `--show-updates` (`-u`) switch, which contacts the repository and adds information about things that are out-of-date:

```
$ svn status --show-updates --verbose
M      *        44        23    sally     README
M               44        20    harry     bar.c
       *        44        35    harry     stuff/trout.c
D               44        19    ira       stuff/fish.c
A                0         ?     ?        stuff/things/bloo.h
Status against revision:   46
```

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

svn diff

[2]Subversion uses its internal diff engine, which produces unified diff format, by default. If you want diff output in a different format, specify an external diff program using `--diff-cmd` and pass any flags you'd like to it using the `--extensions` switch. For example, to see local differences in file `foo.c`

svn revert

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

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

```
$ svn revert README
Reverted 'README'
```

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

```
$ svn status foo
?      foo

$ svn add foo
A         foo

$ svn revert foo
Reverted 'foo'

$ svn status foo
?      foo
```

Note

svn revert *`ITEM`* has exactly the same effect as deleting *`ITEM`* from your working copy and then running **svn update -r BASE** *`ITEM`*. However, if you're reverting a file, **svn revert** has one very noticeable difference—it doesn't have to communicate with the repository to restore your file.

Or perhaps you mistakenly removed a file from version control:

```
$ svn status README
       README

$ svn delete README
D         README

$ svn revert README
Reverted 'README'

$ svn status README
       README
```

Look Ma! No Network!

All three of these commands (**svn status**, **svn diff**, and **svn revert**) can be used without any network access. This makes it easy to manage your changes-in-progress when you are somewhere without a network connection, such as travelling on an airplane, riding a commuter train or hacking on the beach.

Subversion does 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*. This cache (called the “text-base”) also allows Subversion to send the user's local modifications during a commit to the server as a compressed *delta* (or “difference”) against the pristine version. Having this cache is a tremendous benefit—even if you have a fast net connection, it's much faster to send only a file's changes rather than the whole file to the server. At first glance, this might not seem that important, but imagine the repercussions if you try to commit a one line change to a 400MB file and have to send the whole file to the server!

Resolve Conflicts (Merging Others' Changes)

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

```
$ svn update
U  INSTALL
G  README
C  bar.c
Updated to revision 46.
```

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

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

Whenever a conflict occurs, three things typically occur to assist you in noticing and resolving that conflict:

- Subversion prints a `C` during the update, and remembers that the file is in a state of conflict.
- If Subversion considers the file to be of a mergeable type, it places *conflict markers*—special strings of text which delimit the “sides” of the conflict—into the file to visibly demonstrate the overlapping areas. (Subversion uses the `svn:mime-type` property to decide if a file is capable of contextual, line-based merging. See the section called “`svn:mime-type`” to learn more.)
- For every conflicted file, Subversion places up to three extra unversioned files in your working copy:

  `filename.mine`

```
$ cat sandwich.txt
Top piece of bread
Mayonnaise
Lettuce
Tomato
Provolone
<<<<<<< .mine
Salami
Mortadella
Prosciutto
=======
Sauerkraut
Grilled Chicken
>>>>>>> .r2
Creole Mustard
Bottom piece of bread
```

The strings of less-than signs, equal signs, and greater-than signs are conflict markers, and are not part of the actual data in conflict. You generally want to ensure that those are removed from the file before your next commit. The text between the first two sets of markers is composed of the changes you made in the conflicting area:

```
<<<<<<< .mine
Salami
Mortadella
Prosciutto
=======
```

The text between the second and third sets of conflict markers is the text from Sally's commit:

```
=======
Sauerkraut
Grilled Chicken
>>>>>>> .r2
```

Usually you won't want to just delete the conflict markers and Sally's changes—she's going to be awfully surprised when the sandwich arrives and it's not what she wanted. So this is where you pick up the phone or walk across the office and explain to Sally that you can't get sauerkraut from an Italian deli.[4] Once you've agreed on the changes you will check in, edit your file and remove the conflict markers.

```
Top piece of bread
Mayonnaise
Lettuce
Tomato
Provolone
Salami
Mortadella
Prosciutto
Creole Mustard
Bottom piece of bread
```

Now run **svn resolved**, and you're ready to commit your changes:

[4]And if you ask them for it, they may very well ride you out of town on a rail.

```
$ svn resolved sandwich.txt
$ svn commit -m "Go ahead and use my sandwich, discarding Sally's edits."
```

Remember, if you ever get confused while editing the conflicted file, you can always consult the three files that Subversion creates for you in your working copy—including your file as it was before you updated. You can even use a third-party interactive merging tool to examine those three files.

Copying a File Onto Your Working File

If you get a conflict and decide that you want to throw out your changes, you can merely copy one of the temporary files created by Subversion over the file in your working copy:

```
$ svn update
C  sandwich.txt
Updated to revision 2.
$ ls sandwich.*
sandwich.txt  sandwich.txt.mine  sandwich.txt.r2  sandwich.txt.r1
$ cp sandwich.txt.r2 sandwich.txt
$ svn resolved sandwich.txt
```

Punting: Using svn revert

If you get a conflict, and upon examination decide that you want to throw out your changes and start your edits again, just revert your changes:

```
$ svn revert sandwich.txt
Reverted 'sandwich.txt'
$ ls sandwich.*
sandwich.txt
```

Note that when you revert a conflicted file, you don't have to run **svn resolved**.

Now you're ready to check in your changes. Note that **svn resolved**

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

```
$ svn commit --file logmsg
Sending        sandwich.txt
Transmitting file data .
Committed revision 4.
```

If you fail to specify either the `--message` or `--file` switch, then Subversion will automatically launch your favorite editor (see the `editor-cmd` section in the section called “Config”) for composing a log message.

Tip

If you're in your editor writing a commit message and decide that you want to cancel your commit, you can just quit your editor without saving changes. If you've already saved your commit message, simply delete the text and save again.

```
$ svn commit
Waiting for Emacs...Done

Log message unchanged or not specified
a)bort, c)ontinue, e)dit
a
$
```

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 you that one or more of your files is out-of-date:

```
$ svn commit --message "Add another rule"
Sending        rules.txt
svn: Commit failed (details follow):
svn: Out of date: 'rules.txt' in transaction 'g'
```

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

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

Examining History

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

There are several commands that can provide you with historical data from the repository:

svn log

Shows you broad information: log messages with date and author information attached to revisions, and which paths changed in each revision.

svn diff

Shows you the specific details of how a file changed over time.

svn cat

This is used to retrieve any file as it existed in a particular revision number and display it on your screen.

svn list

Displays the files in a directory for any given revision.

svn log

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

```
$ svn log
```

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

```
$ svn log -r 8 -v
------------------------------------------------------------------------
r8 | sally | 2002-07-14 08:15:29 -0500 | 1 line
Changed paths:
M /trunk/code/foo.c
M /trunk/code/bar.h
A /trunk/code/doc/README

Frozzled the sub-space winch.

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

svn log also takes a `--quiet` (`-q` "pristine" copies in the `.svn` area:

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

Comparing Working Copy to Repository

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

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

Comparing Repository to Repository

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

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

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

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

svn cat

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

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

You can also redirect the output directly into a file:

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

You're probably wondering why we don't just use **svn update --revision** to update the file to the older revision. There are a few reasons why we might prefer to use **svn cat**.

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

Sometimes it's easier to look at an older version of a file in its entirety as opposed to just the differences between it and another revision.

svn list

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

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

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

```
$ svn list --verbose http://svn.collab.net/repos/svn
   2755 harry          1331 Jul 28 02:07 README
   2773 sally               Jul 29 15:07 branches/
   2769 sally               Jul 29 12:07 clients/
   2698 harry               Jul 24 18:07 tags/
```

```
   2785 sally              Jul 29 19:07 trunk/
```

The columns tell you the revision at which the file or directory was last modified, the user who modified it, the size if it is a file, the date it was last modified, and the item's name.

A Final Word on History

In addition to all of the above commands, you can use **svn update** and **svn checkout** with the `--revision` switch to take an entire working copy “back in time” [5]:

```
$ svn checkout --revision 1729 # Checks out a new working copy at r1729
…
$ svn update --revision 1729 # Updates an existing working copy to r1729
…
```

Other Useful Commands

While not as frequently used as the commands previously discussed in this chapter, you will occasionally need these commands.

svn cleanup

When Subversion modifies your working copy (or any information within `.svn`), it tries to do so as

[5]See? We told you that Subversion was a time machine.

svn import

The **svn import** command is a quick way to copy an unversioned tree of files into a repository, creating intermediate directories as necessary.

```
$ svnadmin create /usr/local/svn/newrepos
$ svn import mytree file:///usr/local/svn/newrepos/some/project \
             -m "Initial import"
Adding         mytree/foo.c
Adding         mytree/bar.c
Adding         mytree/subdir
Adding         mytree/subdir/quux.h

Committed revision 1.
```

The previous example copied the contents of directory `mytree` under the directory `some/project` in the repository:

```
$ svn list file:///usr/local/svn/newrepos/some/project
bar.c
foo.c
subdir/
```

Note that after the import is finished, the original tree is *not* converted into a working copy. To start working, you still need to **svn checkout** a fresh working copy of the tree.

Summary

Now we've covered most of the Subversion client commands. Notable exceptions are those dealing with branching and merging (see Chapter 4, *Branching and Merging*) and properties (see the section called “Properties”). However, you may want to take a moment to skim through Chapter 9, *Subversion Complete Reference* to get an iexck4o05 Tm [(top5 Tmc.a0 (tret) -271..013tl7.working) -Tsvn/ning

Chapter 4. Branching and Merging

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

Branching is a fundamental part of version control. If you're going to allow Subversion to manage your data, then this is a feature you'll eventually come to depend on. This chapter assumes that you're already familiar with Subversion's basic concepts (Chapter 2, *Basic Concepts*).

What's a Branch?

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

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

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

This is the basic concept of a *branch*—namely, a line of development that exists independently of another line, yet still shares a common history if you look far enough back in time. A branch always begins life as a copy of something, and moves on from there, generating its own history (see Figure 4.1, “Branches of development”).

Figure 4.1. Branches of development

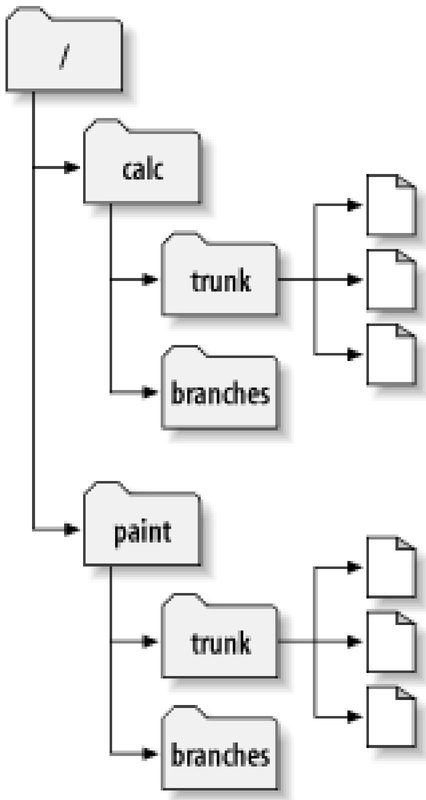
Subversion has commands to help you maintain parallel branches of your files and directories. It allows you to create branches by copying your data, and remembers that the copies are related to one another. It also helps you duplicate changes from one branch to another. Finally, it can make portions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

Using Branches

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

For this chapter, we'll go back to the same example from Chapter 2. Remember that you and your collaborator, Sally, are sharing a repository that contains two projects, `paint` and `calc`. Notice that in Figure 4.2, “Starting repository layout”, however, each project directory now contains subdirectories named `trunk` and `branches`. The reason for this will soon become clear.

Figure 4.2. Starting repository layout



As before, assume that Sally and you both have working copies of the “calc” project. Specifically, you each have a working copy of `/calc/trunk`. All the files for the project are in this subdirectory rather than in `/calc` itself, because your team has decided that `/calc/trunk` is where the “main line” of development is going to take place.

Let's say that you've been given the task of performing a radical reorganization of the project. It will take a long time to write, and will affect all the files in the project. The problem here is that you don't want to interfere with Sally, who is in the process of fixing small bugs here and there. She's depending on the fact that the latest version of the project (in `/calc/trunk`) is always usable. If you start committing your changes bit-by-bit, you'll surely break things for Sally.

One strategy is to crawl into a hole: you and Sally can stop sharing information for a week or two. That is, start gutting and reorganizing all the files in your working copy, but don't commit or update until you're completely finished with the task. There are a number of problems with this, though. First, it's not very safe. Most people like to save their work to the repository frequently, should something bad accidentally happen to their working copy. Second, it's not very flexible. If you do your work on different computers (perhaps you have a working copy of `/calc/trunk` on two different machines), you'll need to manually copy your changes back and forth, or just do all the work on a single computer. By that same token, it's difficult to share your changes-in-progress with anyone else. A common software development "best practice" is to allow your peers to review your work as you go. If nobody sees your intermediate commits, you lose potential feedback. Finally, when you're finished with all your changes, you might find it very difficult to re-merge your final work with the rest of the company's main body of code. Sally (or others) may have made many other changes in the repository that are difficult to incorporate into your working copy—especially if you run **svn update** after weeks of isolation.

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

Creating a Branch

Creating a branch is very simple—you make a copy of the project in the repository using the **svn copy** command. Subversion is not only able to copy single files, but whole directories as well. In this case, you want to make a copy of the `/calc/trunk` directory. Where should the new copy live? Wherever you wish—it's a matter of project policy. Let's say that your team has a policy of creating branches in the `/calc/branches` area of the repository, and you want to name your branch `my-calc-branch`. You'll want to create a new directory, `/calc/branches/my-calc-branch`, which begins its life as a copy of `/calc/trunk`.

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

```
$ svn checkout http://svn.example.com/repos/calc bigwc
A  bigwc/trunk/
A  bigwc/trunk/Makefile
A  bigwc/trunk/integer.c
A  bigwc/trunk/button.c
A  bigwc/branches/
Checked out revision 340.
```

Making a copy is now simply a matter of passing two working-copy paths to the **svn copy** command:

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

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

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

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

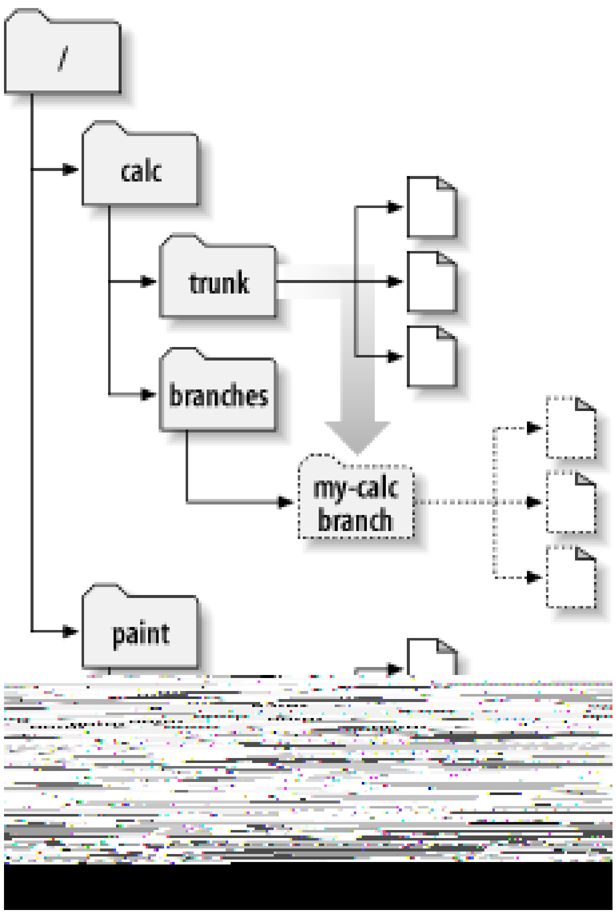
```
$ svn copy http://svn.example.com/repos/calc/trunk \
```

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

Committed revision 341.
```

There's really no difference between these two methods. Both procedures create a new directory in revision 341, and the new directory is a copy of `/calc/trunk`. This is shown in Figure 4.3, "Repository with new copy". Notice that the second method, however, performs an *immediate* commit. [1] It's an easier procedure, because it doesn't require you to check out a large mirror of the repository. In fact, this technique doesn't even require you to have a working copy at all.

Figure 4.3. Repository with new copy



> **Cheap Copies**
>
> Subversion's repository has a special design. When you copy a directory, you don't need to worry about the repository growing huge—Subversion doesn't actually duplicate any data. Instead, it creates a new directory entry that points to an *existing* tree. If you're a Unix user, this is the same concept as a hard-link. From there, the copy is said to be "lazy". That is, if you commit a change to one file within the copied directory, then only that file changes—the rest of the files continue to exist as links to the original files in the original directory.

[1] Subversion does not support cross-repository copying. When using URLs with **svn copy** or **svn move**, you can only copy items within the same repository.

This is why you'll often hear Subversion users talk about “cheap copies”. It doesn't matter how large the directory is—it takes a very tiny, constant amount of time to make a copy of it. In fact, this feature is the basis of how commits work in Subversion: each revision is a “cheap copy” of the previous revision, with a few items lazily changed within. (To read more about this, visit Subversion's website and read about the “bubble up” method in Subversion's design documents.)

Of course, these internal mechanics of copying and sharing data are hidden from the user, who simply sees copies of trees. The main point here is that copies are cheap, both in time and space. Make branches as often as you want.

Working with Your Branch

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

```
$ svn checkout http://svn.example.com/repos/calc/branches/my-calc-branch
A  my-calc-branch/Makefile
A  my-calc-branch/integer.c
A  my-calc-branch/button.c
Checked out revision 341.
```

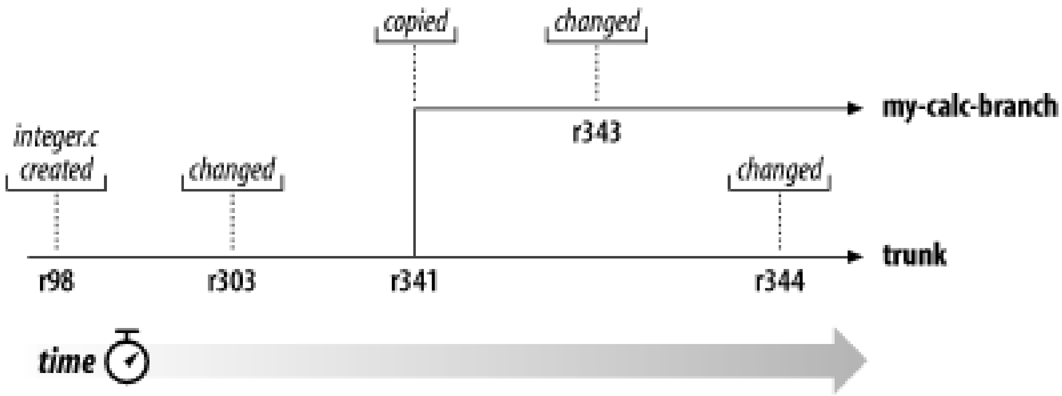
There's nothing special about this working copy; it simply mirrors a different directory in the repository. When you commit changes, however, Sally won't ever see them when she updates. Her working copy is of `/calc/trunk`. (Be sure to read the section called “Switching a Working Copy” later in this chapter: the **svn switch** command is an alternate way of creating a working copy of a branch.)

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

- You make a change to `/calc/branches/my-calc-branch/button.c`, which creates revision 342.
- You make a change to `/calc/branches/my-calc-branch/integer.c`, which creates revision 343.
- Sally makes a change to `/calc/trunk/integer.c`, which creates revision 344.

There are now two independent lines of development, shown in Figure 4.4, “The branching of one file's history”, happening on `integer.c`.

Figure 4.4. The branching of one file's history



copied
changed
my-calc-branch
r343
integer.c
created
changed
changed
trunk
r98
r303
r341
r344
time

```
$ svn log --verbose integer.c
------------------------------------------------------------------------
r344 | sally | 2002-11-07 15:27:56 -0600 (Thu, 07 Nov 2002) | 2 lines
Changed paths:
   M /calc/trunk/integer.c

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

------------------------------------------------------------------------
r303 | sally | 2002-10-29 21:14:35 -0600 (Tue, 29 Oct 2002) | 2 lines
Changed paths:
   M /calc/trunk/integer.c

* integer.c:  changed a docstring.

------------------------------------------------------------------------
r98 | sally | 2002-02-22 15:35:29 -0600 (Fri, 22 Feb 2002) | 2 lines
Changed paths:
   M /calc/trunk/integer.c

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

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

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

The Key Concepts Behind Branches

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

1. Unlike many other version control systems, Subversion's branches exist as *normal filesystem directories* in the repository, not in an extra dimension. These directories just happen to carry some extra historical information.

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

Copying Changes Between Branches

Now you and Sally are working on parallel branches of the project: you're working on a private branch, and Sally is working on the *trunk*, or main line of development.

For projects that have a large number of contributors, it's common for most people to have working copies of the trunk. Whenever someone needs to make a long-running change that is likely to disrupt the trunk, a standard procedure is to create a private branch and commit changes there until all the work is complete.

So, the good news is that you and Sally aren't interfering with each other. The bad news is that it's very easy to drift *too* far apart. Remember that one of the problems with the “crawl in a hole” strategy is that by the time you're finished with your branch, it may be near-impossible to merge your changes back into the trunk without a huge number of conflicts.

Instead, you and Sally might continue to share changes as you work. It's up to you to decide which changes are worth sharing; Subversion gives you the ability to selectively “copy” changes between branches. And when you're completely finished with your branch, your entire set of branch changes can be copied back into the trunk.

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

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

```
$ svn merge -r 343:344 http://svn.example.com/repos/calc/trunk
U  integer.c

$ svn status
M  integer.c
```

The output of **svn merge** shows that your copy of `integer.c` was patched. It now contains Sally's change—the change has been “copied” from the trunk to your working copy of your private branch, and now exists as a local modification. At this point, it's up to you to review the local modification and make sure it works correctly.

In another scenario, it's possible that things may not have gone so well, and that `integer.c` may have entered a conflicted state. You might need to resolve the conflict using standard procedures (see Chapter 3), or if you decide that the merge was a bad idea altogether, simply give up and **svn revert** the local change.

But assuming that you've reviewed the merged change, you can **svn commit** the change as usual. At that point, the change has been merged into your repository branch. In version control terminology, this act of copying changes between branches is commonly called *porting* changes.

When you commit the local modification, make sure your log message mentions that you're porting a change had, say, added a new directory, the output of **svn diff** wouldn't have mentioned it at all. **svn diff** only outputs the limited patch-format, so there are some ideas it simply can't express. [2] The **svn merge** command, however, can express changes in tree structure and properties by directly applying them to your working copy.

A word of warning: while **svn diff** and **svn merge** are very similar in concept, they do have different syntax in many cases. Be sure to read about them in Chapter 9 for details, or ask **svn help**. For example, **svn merge** requires a working-copy path as a target, i.e. a place where it should apply the tree-changes. If the target isn't specified, it assumes you are trying to perform one of the following common operations:

1. You want to merge directory changes into your current working directory.
2. You want to merge the changes in a specific file into a file by the same name which exists in your current working directory.

If you are merging a directory and haven't specified a target path, **svn merge** assumes the first case above and tries to apply the changes into your current directory. If you are merging a file, and that file (or a file by the same name) exists in your current working directory, **svn merge** assumes the second case and tries to apply the changes to a local file with the same name.

If you want changes applied somewhere else, you'll need to say so. For example, if you're sitting in the parent directory of your working copy, you'll have to specify the target directory to receive the changes:

```
$ svn merge -r 343:344 http://svn.example.com/repos/calc/trunk my-calc-branch
U   my-calc-branch/integer.c
```

The Key Concept Behind Merging

You've now seen an example of the **svn merge** command, and you're about to see several more. If you're feeling confused about exactly how merging works, you're not alone. Many users (especially those new to version control) are initially perplexed about the proper syntax of the command, and about how and when the feature should be used. But fear not, this command is actually much simpler than you think! There's a very easy technique for understanding exactly how **svn merge** behaves.

The main source of confusion is the *name* of the command. The term “merge” somehow denotes that branches are combined together, or that there's some sort of mysterious blending of data going on. That's not the case. A better name for the command might have been **svn diff-and-apply**, because that's all that happens: two repository trees are compared, and the differences are applied to a working copy.

The command takes three arguments:

1. An initial repository tree (often called the *left side* of the comparison),
2. A final repository tree (often called the *right side* of the comparison),
3. A working copy to accept the differences as local changes (often called the *target* of the merge).

Once these three arguments are specified, the two trees are compared, and the resulting differences are applied to the target working copy as local modifications. When the command is done, the results are no

[2] In the future, the Subversion project plans to use (or invent) an expanded patch format that describes changes in tree structure and properties.

different than if you had hand-edited the files, or run various **svn add** or **svn delete**

In cases like this, people take comfort in being able to predict or examine merges before they happen. One simple way to do that is to run **svn diff** with the same arguments you plan to pass to **svn merge**, as we already showed in our first example of merging. Another method of previewing is to pass the `--dry-run` option to the merge command:

```
$ svn merge --dry-run -r 343:344 http://svn.example.com/repos/calc/trunk
U  integer.c

$ svn status
#  nothing printed, working copy is still unchanged.
```

The `--dry-run` option doesn't actually apply any local changes to the working copy. It only shows status codes that *would* be printed in a real merge. It's useful for getting a “high level” preview of the potential merge, for those times when running **svn diff** gives too much detail.

Subversion and Changesets

Everyone seems to have a slightly different definition of “changeset”, or at least a different expectation of what it means for a version control system to have “changeset features”. For our purpose, let's say that a changeset is just a collection of changes with a unique name. The changes might include textual edits to file contents, modifications to tree structure, or tweaks to metadata. In more common speak, a changeset is just a patch with a name you can refer to.

In Subversion, a global revision number N names a tree in the repository: it's the way the repository looked after the Nth commit. It's also the name of an implicit changeset: if you compare tree N with tree N-1, you can derive the exact patch that was committed. For this reason, it's easy to think of “revision N” as not just a tree, but a changeset as well. If you use an issue tracker to manage bugs, you can use the revision numbers to refer to particular patches that fix bugs—for example, “this issue was fixed by revision 9238.”. Somebody can then run **svn log -r9238** to read about the exact changeset which fixed the bug, and run **svn diff -r9237:9238** to see the patch itself. And Subversion's `merge` command also uses revision numbers. You can merge specific changesets from one branch to another by naming them in the merge arguments: **svn merge -r9237:9238** would merge changeset #9238 into your working copy.

Merge Conflicts

Just like the **svn update** command, **svn merge** applies changes to your working copy. And therefore it's also capable of creating conflicts. The conflicts produced by **svn merge**, however, are sometimes different, and this section explains those differences.

To begin with, assume that your working copy has no local edits. When you **svn update** to a particular revision, the changes sent by the server will always apply “cleanly” to your working copy. The server produces the delta by comparing two trees: a virtual snapshot of your working copy, and the revision tree you're interested in. Because the left-hand side of the comparison is exactly equal to what you already have, the delta is guaranteed to correctly convert your working copy into the right-hand tree.

But **svn merge** has no such guarantees and can be much more chaotic: the user can ask the server to compare *any* two trees at all, even ones that are unrelated to the working copy! This means there's large potential for human error. Users will sometimes compare the wrong two trees, creating a delta that doesn't apply cleanly. **svn merge** will do its best to apply as much of the delta as possible, but some parts may be impossible. Just like the Unix **patch** command sometimes complains about “failed hunks”, **svn merge** will complain about “skipped targets”:

```
$ svn merge -r 1288:1351 http://svn.example.com/repos/branch
U  foo.c
```

```
U  bar.c
Skipped missing target: 'baz.c'
U  glub.c
C  glorb.h

$
```

In the previous example it might be the case that `baz.c` exists in both snapshots of the branch being compared, and the resulting delta wants to change the file's contents, but the file doesn't exist in the working copy. Whatever the case, the “skipped” message means that the user is most likely comparing the wrong two trees; they're the classic sign of driver error. When this happens, it's easy to recursively revert all the changes created by the merge (**svn revert --recursive**), delete any unversioned files or directories left behind after the revert, and re-run **svn merge** with different arguments.

Also notice that the previous example shows a conflict happening on `glorb.h`. We already stated that the working copy has no local edits: how can a conflict possibly happen? Again, because the user can use **svn merge** to define and apply any old delta to the working copy, that delta may contain textual changes that don't cleanly apply to a working file, even if the file has no local modifications.

Another small difference between **svn update** and **svn merge** are the names of the full-text files created when a conflict happens. In the section called “Resolve Conflicts (Merging Others' Changes)”, we saw that an update produces files named `filename.mine`, `filename.rOLDREV`, and `filename.rNEWREV`. When **svn merge** produces a conflict, though, it creates three files named `filename.working`, `filename.left`, and `filename.right`. In this case, the terms “left” and “right” are describing which side of the double-tree comparison the file came from. In any case, these

In these situations, you'll want **svn merge** to do a path-based comparison only, ignoring any relations between files and directories. Add the `--ignore-ancestry` option to your merge command, and it will behave just like **svn diff**. (And conversely, the `--notice-ancestry` option will cause **svn diff** to behave like the `merge` command.)

Common Use-Cases

There are many different uses for branching and **svn merge**, and this section describes the most common ones you're likely to run into.

Merging a Whole Branch to Another

To complete our running example, we'll move forward in time. Suppose several days have passed, and many changes have happened on both the trunk and your private branch. Suppose that you've finished working on your private branch; the feature or bug fix is finally complete, and now you want to merge all of your branch changes back into the trunk for others to enjoy.

So how do we use **svn merge** in this scenario? Remember that this command compares two trees, and applies the differences to a working copy. So to receive the changes, you need to have a working copy of the trunk. We'll assume that either you still have your original one lying around (fully updated), or that you recently checked out a fresh working copy of `/calc/trunk`.

But which two trees should be compared? At first glance, the answer may seem obvious: just compare the latest trunk tree with your latest branch tree. But beware—this assumption is *wrong*, and has burned many a new user! Since **svn merge** operates like **svn diff**, comparing the latest trunk and branch trees will *not* merely describe the set of changes you made to your branch. Such a comparison shows too many changes: it would not only show the addition of your branch changes, but also the *removal* of trunk changes that never happened on your branch.

To express only the changes that happened on your branch, you need to compare the initial state of your branch to its final state. Using **svn log** on your branch, you can see that your branch was created in revision 341. And the final state of your branch is simply a matter of using the

As expected, the final revision printed by this command is the revision in which `my-calc-branch` was created by copying.

Here's the final merging procedure, then:

```
$ cd calc/trunk
$ svn update
At revision 405.

$ svn merge -r 341:405 http://svn.example.com/repos/calc/branches/my-calc-branch
U   integer.c
U   button.c
U   Makefile

$ svn status
M   integer.c
M   button.c
M   Makefile

# ...examine the diffs, compile, test, etc...

$ svn commit -m "Merged my-calc-branch changes r341:405 into the trunk."
Sending        integer.c
Sending        button.c
Sending        Makefile
Transmitting file data ...
Committed revision 406.
```

Again, notice that the commit log message very specifically mentions the range of changes that was merged into the trunk. Always remember to do this, because it's critical information you'll need later on.

For example, suppose you decide to keep working on your branch for another week, in order to complete an enhancement to your original feature or bug fix. The repository's `HEAD` revision is now 480, and you're ready to do another merge from your private branch to the trunk. But as discussed in the section called “Best Practices for Merging”, you don't want to merge the changes you've already merged before; you only want to merge everything “new” on your branch since the last time you merged. The trick is to figure out what's new.

The first step is to run **svn log** on the trunk, and look for a log message about the last time you merged from the branch:

```
$ cd calc/trunk
$ svn log
…
------------------------------------------------------------------------
r406 | user | 2004-02-08 11:17:26 -0600 (Sun, 08 Feb 2004) | 1 line

Merged my-calc-branch changes r341:405 into the trunk.
------------------------------------------------------------------------
…
```

Aha! Since all branch-changes that happened between revisions 341 and 405 were previously merged to the trunk as revision 406, you now know that you want to merge only the branch changes after that—by comparing revisions 406 and `HEAD`.

```
$ cd calc/trunk
```

```
$ svn update
At revision 480.

# We notice that HEAD is currently 480, so we use it to do the merge:

$ svn merge -r 406:480 http://svn.example.com/repos/calc/branches/my-calc-branch
U   integer.c
U   button.c
U   Makefile

$ svn commit -m "Merged my-calc-branch changes r406:480 into the trunk."
Sending        integer.c
Sending        button.c
Sending        Makefile
Transmitting file data ...
Committed revision 481.
```

Now the trunk contains the complete second wave of changes made to the branch. At this point, you can either delete your branch (we'll discuss this later on), or continue working on your branch and repeat this procedure for subsequent merges.

Undoing Changes

Another common use for **svn merge** is to roll back a change that has already been committed. Suppose you're working away happily on a working copy of `/calc/trunk`, and you discover that the change made way back in revision 303, which changed `integer.c`, is completely wrong. It never should have been committed. You can use **svn merge** to "undo" the change in your working copy, and then commit the local modification to the repository. All you need to do is to specify a *reverse* difference:

```
$ svn merge -r 303:302 http://svn.example.com/repos/calc/trunk
U  integer.c

$ svn status
M  integer.c

$ svn diff
…
# verify that the change is removed
…

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

One way to think about a repository revision is as a specific group of changes (some version control systems call these *changesets*). By using the `-r` switch, you can ask **svn merge** to apply a changeset, or whole range of changesets, to your working copy. In our case of undoing a change, we're asking **svn merge** to apply changeset #303 to our working copy *backwards*.

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

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

Yes, that's true. When we talk about “removing” a change, we're really talking about removing it from HEAD. The original change still exists in the repository's history. For most situations, this is good enough. Most people are only interested in tracking the HEAD of a project anyway. There are special cases, however, where you really might want to destroy all evidence of the commit. (Perhaps somebody accidentally committed a confidential document.) This isn't so easy, it turns out, because Subversion was deliberately designed to never lose information. Revisions are immutable trees which build upon one another. Removing a revision from history would cause a domino effect, creating chaos in all subsequent revisions and possibly invalidating all working copies. [3]

Resurrecting Deleted Items

The great thing about version control systems is that information is never lost. Even when you delete a file or directory, it may be gone from the HEAD revision, but the object still exists in earlier revisions. One of the most common questions new users ask is, “How do I get my old file or directory back?”.

The first step is to define exactly **which** item you're trying to resurrect. Here's a useful metaphor: you can think of every object in the repository as existing in a sort of two-dimensional coordinate system. The first coordinate is a particular revision tree, and the second coordinate is a path within that tree. So every version of your file or directory can be defined by a specific coordinate pair.

Subversion has no Attic directory like CVS does, [4] so you need to use **svn log** to discover the exact coordinate pair you wish to resurrect. A good strategy is to run **svn log --verbose** in a directory which used to contain your deleted item. The --verbose option shows a list of all changed items in each revision; all you need to do is find the revision in which you deleted the file or directory. You can do this visually, or by using another tool to examine the log output (via **grep**, or perhaps via an incremental search in an editor).

```
$ cd parent-dir
$ svn log --verbose
…
------------------------------------------------------------------------
r808 | joe | 2003-12-26 14:29:40 -0600 (Fri, 26 Dec 2003) | 3 lines
Changed paths:
   D /calc/trunk/real.c
   M /calc/trunk/integer.c

Added fast fourier transform functions to integer.c.
Removed real.c because code now in double.c.
…
```

In the example, we're assuming that you're looking for a deleted file real.c. By looking through the logs of a parent directory, you've spotted that this file was deleted in revision 808. Therefore, the last versionlogs oflogs of

[3]The Subversion project has plans, however, to someday implement an **svnadmin obliterate** command that would accomplish the task of permanently deleting information. In the meantime, see the section called “svndumpfilter” for a possible workaround.
[4]Because CVS doesn't version trees, it creates an Attic area within each repository directory as a way of remembering deleted files.

In this particular example, however, this is probably not the best strategy. Reverse-applying revision 808 would not only schedule `real.c` for addition, but the log message indicates that it would also undo certain changes to `integer.c`, which you don't want. Certainly, you could reverse-merge revision 808 and then **svn revert** the local modifications to `integer.c`, but this technique doesn't scale well. What if there were 90 files changed in revision 808?

A second, more targeted strategy is not to use **svn merge** at all, but rather the **svn copy** command. Simply copy the exact revision and path "coordinate pair" from the repository to your working copy:

```
$ svn copy --revision 807 \
           http://svn.example.com/repos/calc/trunk/real.c ./real.c

$ svn status
A  +   real.c

$ svn commit -m "Resurrected real.c from revision 807, /calc/trunk/real.c."
Adding         real.c
Transmitting file data .
Committed revision 1390.
```

The plus sign in the status output indicates that the item isn't merely scheduled for addition, but scheduled for addition "with history". Subversion remembers where it was copied from. In the future, running **svn log** on this file will traverse back through the file's resurrection and through all the history it had prior to revision 807. In other words, this new `real.c` isn't really new; it's a direct descendant of the original, deleted file.

Although our example shows us resurrecting a file, note that these same techniques work just as well for resurrecting deleted directories.

Common Branching Patterns

Version control is most often used for software development, so here's a quick peek at two of the most common branching/merging patterns used by teams of programmers. If you're not using Subversion for software development, feel free to skip this section. If you're a software developer using version control for the first time, pay close attention, as these patterns are often considered best practices by experienced folk. These processes aren't specific to Subversion; they're applicable to any version control system. Still, it may help to see them described in Subversion terms.

Release Branches

Most software has a typical lifecycle: code, test, release, repeat. There are two problems with this process. First, developers need to keep writing new features while quality-assurance teams take time to test supposedly-stable versions of the software. New work cannot halt while the software is tested. Second, the team almost always needs to support older, released versions of software; if a bug is discovered in the latest code, it most likely exists in released versions as well, and customers will want to get that bugfix without having to wait for a major new release.

Here's where version control can help. The typical procedure looks like this:

- *Developers commit all new work to the trunk.* Day-to-day changes are committed to `/trunk`: new features, bugfixes, and so on.

- *The trunk is copied to a "release" branch.* When the team thinks the software is ready for release (say, a 1.0 release), then `/trunk` might be copied to `/branches/1.0`.

- *Teams continue to work in parallel.* One team begins rigorous testing of the release branch, while another team continues new work (say, for version 2.0) on `/trunk`. If bugs are discovered in either

```
$ cd trunk-working-copy

$ svn update
At revision 1910.

$ svn merge http://svn.example.com/repos/calc/trunk@1910 \
            http://svn.example.com/repos/calc/branches/mybranch@1910
U  real.c
U  integer.c
A  newdirectory
A  newdirectory/newfile
…
```

1. Copy the project's entire “trunk” to a new branch directory.

2. Switch only *part* of the trunk working copy to mirror the branch.

In other words, if a user knows that the branch-work only needs to happen on a specific subdirectory, they use **svn switch** to move only that subdirectory to the branch. (Or sometimes users will switch just a single working file to the branch!) That way, they can continue to receive normal “trunk” updates to most of their working copy, but the switched portions will remain immune (unless someone commits a change to their branch). This feature adds a whole new dimension to the concept of a “mixed working copy”—not only can working copies contain a mixture of working revisions, but a mixture of repository locations as well.

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

Note that while it's okay for your working copy to reflect a mixture of repository locations, these locations must all be within the *same* repository. Subversion repositories aren't yet able to communicate with one another; that's a feature planned beyond Subversion 1.0.[5]

Switches and Updates

Have you noticed that the output of **svn switch** and **svn update** look the same? The `switch` command is actually a superset of the update command.

When you run **svn update**, you're asking the repository to compare two trees. The repository does so, and then sends a description of the differences back to the client. The only difference between **svn switch** and **svn update** is that the `update` command always compares two identical paths.

That is, if your working copy is a mirror of `/calc/trunk`, then **svn update** will automatically compare

[5]You *can*, however, use **svn switch** with the `--relocate` switch if the URL of your server changes and you don't want to abandon an existing working copy. See the **svn switch** section in Chapter 9, *Subversion Complete Reference* for more information and an example.

version, this idea already seems to be everywhere. Each repository revision is exactly that—a snapshot of the filesystem after each commit.

However, people often want to give more human-friendly names to tags, like `release-1.0`. And they want to make snapshots of smaller subdirectories of the filesystem. After all, it's not so easy to remember that release-1.0 of a piece of software is a particular subdirectory of revision 4822.

Creating a Simple Tag

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

```
$ svn copy http://svn.example.com/repos/calc/trunk \
           http://svn.example.com/repos/calc/tags/release-1.0 \
      -m "Tagging the 1.0 release of the 'calc' project."

Committed revision 351.
```

This example assumes that a `/calc/tags` directory already exists. (If it doesn't, see svn mkdir). After the copy completes, the new `release-1.0` directory is forever a snapshot of how the project looked in the `HEAD` revision at the time you made the copy. Of course you might want to be more precise about exactly which revision you copy, in case somebody else may have committed changes to the project when you weren't looking. So if you know that revision 350 of `/calc/trunk` is exactly the snapshot you want, you can specify it by passing `-r 350` to the **svn copy** command.

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

If you are administering a repository, there are two approaches you can take to managing tags. The first approach is “hands off”: as a matter of project policy, decide where your tags will live, and make sure all users know how to treat the directories they copy in there. (That is, make sure they know not to commit to them.) The second approach is more paranoid: you can use one of the access-control scripts provided with Subversion to prevent anyone from doing anything but creating new copies in the tags-area (See Chapter 6, *Server Configuration*.) The paranoid approach, however, isn't usually necessary. If a user accidentally commits a change to a tag-directory, you can simply undo the change as discussed in the previous section. This is version control, after all.

Creating a Complex Tag

Sometimes you may want your “snapshot” to be more complicated than a single directory at a single revision.

For example, pretend your project is much larger than our `calc` example: suppose it contains a number of subdirectories and many more files. In the course of your work, you may decide that you need to create a working copy that is designed to have specific features and bug fixes. You can accomplish this by selectively backdating files or directories to particular revisions (using **svn update -r** liberally), or by switching files and directories to particular branches (making use of **svn switch**). When you're done, your working copy is a hodgepodge of repository locations from different revisions. But after testing, you know it's the precise combination of data you need.

Time to make a snapshot. Copying one URL to another won't work here. In this case, you want to make a snapshot of your exact working copy arrangement and store it in the repository. Luckily, **svn copy** actually has four different uses (which you can read about in Chapter 9), including the ability to copy a working-copy tree to the repository:

```
$ ls
my-working-copy/

$ svn copy my-working-copy http://svn.example.com/repos/calc/tags/mytag

Committed revision 352.
```

Now there is a new directory in the repository, `/calc/tags/mytag`, which is an exact snapshot of your working copy—mixed revisions, URLs, and all.

Other users have found interesting uses for this feature. Sometimes there are situations where you have a bunch of local changes made to your working copy, and you'd like a collaborator to see them. Instead of running **svn diff** and sending a patch file (which won't capture tree changes, symlink changes or changes in properties), you can instead use **svn copy** to "upload" your working copy to a private area of the repository. Your collaborator can then either checkout a verbatim copy of your working copy, or use **svn merge** to receive your exact changes.

Branch Maintenance

You may have noticed by now that Subversion is extremely flexible. Because it implements branches and tags with the same underlying mechanism (directory copies), and because branches and tags appear in normal filesystem space, many people find Subversion intimidating. It's almost *too* flexible. In this section, we'll offer some suggestions for arranging and managing your data over time.

Repository Layout

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

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

If a repository contains multiple projects, admins typically index their layout by project (see the section called "Choosing a Repository Layout" to read more about "project roots"):

```
/paint/trunk
/paint/branches
/paint/tags
/calc/trunk
/calc/branches
/calc/tags
```

Of course, you're free to ignore these common layouts. You can create any sort of variation, whatever works best for you or your team. Remember that whatever you choose, it's not a permanent commitment. You can reorganize your repository at rep brlb-(/cn) -331.5 (brlb-(/cn) 5mmtory) y2ggt,u

Remember, though, that while moving directories may be easy to do, you need to be considerate of your users as well. Your juggling can be disorienting to users with existing working copies. If a user has a working copy of a particular repository directory, your **svn move** operation might remove the path from the latest revision. When the user next runs **svn update**, she will be told that her working copy represents a path that no longer exists, and the user will be forced to **svn switch** to the new location.

Data Lifetimes

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

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

Committed revision 375.
```

And now your branch is gone. Of course it's not really gone: the directory is simply missing from the `HEAD` revision, no longer distracting anyone. If you use **svn checkout**, **svn switch**, or **svn list** to examine an earlier revision, you'll still be able to see your old branch.

If browsing your deleted directory isn't enough, you can always bring it back. Resurrecting data is very easy in Subversion. If there's a deleted directory (or file) that you'd like to bring back into `HEAD`, simply use **svn copy -r** to copy it from the old revision:

```
$ svn copy -r 374 http://svn.example.com/repos/calc/branches/my-calc-branch \
                  http://svn.example.com/repos/calc/branches/my-calc-branch

Committed revision 376.
```

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

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

Committed revision 377.
```

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

Summary

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

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

Chapter 5. Repository Administration

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

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

If you plan to access a Subversion repository only in the role of a user whose data is under version control (that is, via a Subversion client), you can skip this chapter altogether. However, if you are, or wish to become, a Subversion repository administrator, [1] you should definitely pay attention to this chapter.

Repository Basics

Before jumping into the broader topic of repository administration, let's further define what a repository is. How does it look? How does it feel? Does it take its tea hot or iced, sweetened, and with lemon? As an administrator, you'll be expected to understand the composition of a repository both from a logical perspective—dealing with how data is represented inside the repository—and from a physical nuts-and-bolts perspective—how a repository looks and acts with respect to non-Subversion tools. The following section covers some of these basic concepts at a very high level.

Understanding Transactions and Revisions

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

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

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

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

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

In theory, someday whole workflow applications might revolve around more fine-grained control of

[2] Pronounced “fuzz-fuzz”, if Jack Repenning has anything to say about it.

Feature	Berkeley DB	FSFS
Platform-independent storage	no	yes
Usable over network filesystems	no	yes
Repository size	slightly larger	slightly smaller
Scalability: number of revision trees	database; no problems	some older native filesystems don't scale well with thousands of entries in a single directory.
Scalability: directories with many files	slower	faster
Speed: checking out latest code	faster	slower
Speed: large commits	slower, but work is spread throughout commit	faster, but finalization delay may cause client timeouts
Group permissions handling	sensitive to user umask problems; best if accessed by only one user.	works around umask problems
Code maturity	in use since 2001	in use since 2004

Berkeley DB

When the initial design phase of Subversion was in progress, the developers decided to use Berkeley DB for a variety of reasons, including its open-source license, transaction support, reliability, performance, API simplicity, thread-safety, support for cursors, and so on.

Berkeley DB provides real transaction support—perhaps its most powerful feature. Multiple processes accessing your Subversion repositories don't have to worry about accidentally clobbering each other's data. The isolation provided by the transaction system is such that for any given operation, the Subversion repository code sees a static view of the database—not a database that is constantly changing at the hand of some other process—and can make decisions based on that view. If the decision made happens to conflict with what another process is doing, the entire operation is rolled back as if it never happened, and Subversion gracefully retries the operation against a new, updated (and yet still static) view of the database.

Another great feature of Berkeley DB is *hot backups*—the ability to backup the database environment without taking it “offline”. We'll discuss how to backup your repository in the section called “Repository Backup”, but the benefits of being able to make fully functional copies of your repositories without any downtime should be obvious.

Berkeley DB is also a very reliable database system. Subversion uses Berkeley DB's logging facilities, which means that the database first writes to on-disk log files a description of any modifications it is about to make, and then makes the modification itself. This is to ensure that if anything goes wrong, the database system can back up to a previous *checkpoint*—a location in the log files known not to be corrupt—and replay transactions until the data is restored to a usable state. See the section called “Managing Disk Space” for more about Berkeley DB log files.

But every rose has its thorn, and so we must note some known limitations of Berkeley DB. First, Berkeley DB environments are not portable. You cannot simply copy a Subversion repository that was created on a Unix system onto a Windows system and expect it to work. While much of the Berkeley DB database format is architecture independent, there are other aspects of the environment that are not. Secondly, Subversion uses Berkeley DB in a way that will not operate on Windows 95/98 systems—if you need to house a repository on a Windows machine, stick with Windows 2000 or Windows XP. Also, you should never keep a Berkeley DB repository on a network share. While Berkeley DB promises to behave correctly on network shares that meet a particular set of specifications, almost no known shares actually meet all those specifications.

Finally, because Berkeley DB is a library linked directly into Subversion, it's more sensitive to interruptions than a typical relational database system. Most SQL systems, for example, have a dedicated server process that mediates all access to tables. If a program accessing the database crashes for some reason, the database daemon notices the lost connection and cleans up any mess left behind. And because the database daemon is the only process accessing the tables, applications don't need to worry about permission conflicts. These things are not the case with Berkeley DB, however. Subversion (and programs using Subversion libraries) access the database tables directly, which means that a program crash can leave the database in a temporarily inconsistent, inaccessible state. When this happens, an administrator needs to ask Berkeley DB to restore to a checkpoint, which is a bit of an annoyance. Other things can cause a repository to “wedge” besides crashed processes, such as programs conflicting over ownership and permissions on the database files. So while a Berkeley DB repository is quite fast and scalable, it's best used by a single server process running as one user—such as Apache's **httpd** or **svnserve** (see Chapter 6, *Server Configuration*)—rather than accessing it as many different users via `file:///` or `svn+ssh://` URLs. If using a Berkeley DB repository directly as multiple users, be sure to read the section called “Supporting Multiple Repository Access Methods”.

FSFS

In mid-2004, a second type of repository storage system came into being: one which doesn't use a database at all. An FSFS repository stores a revision tree in a single file, and so all of a repository's revisions can be found in a single subdirectory full of numbered files. Transactions are created in separate subdirectories. When complete, a single transaction file is created and moved to the revisions directory, thus guaranteeing that commits are atomic. And because a revision file is permanent and unchanging, the repository also can be backed up while “hot”, just like a Berkeley DB repository.

The revision-file format represents a revision's directory structure, file contents, and deltas against files in other revision trees. Unlike a Berkeley DB database, this storage format is portable across different operating systems and isn't sensitive to CPU architecture. Because there's no journaling or shared-memory files being used, the repository can be safely accessed over a network filesystem and examined in a read-only environment. The lack of database overhead also means that the overall repository size is a bit smaller.

FSFS has different performance characteristics too. When committing a directory with a huge number of files, FSFS uses an O(N) algorithm to append entries, while Berkeley DB uses an O(N^2) algorithm to rewrite the whole directory. On the other hand, FSFS writes the latest version of a file as a delta against an earlier version, which means that checking out the latest tree is a bit slower than fetching the fulltexts stored in a Berkeley DB HEAD revision. FSFS also has a longer delay when finalizing a commit, which could in extreme cases cause clients to time out when waiting for a response.

The most important distinction, however, is FSFS's inability to be “wedged” when something goes wrong. If a process using a Berkeley DB database runs into a permissions problem or suddenly crashes, the database is left unusable until an administrator recovers it. If the same scenarios happen to a process using an FSFS repository, the repository isn't affected at all. At worst, some transaction data is left behind.

The only real argument against FSFS is its relative immaturity compared to Berkeley DB. It hasn't been used or stress-tested nearly as much, and so a lot of these assertions about speed and scalability are just that: assertions, based on good guesses. In theory, it promises a lower barrier to entry for new administrators and is less susceptible to problems. In practice, only time will tell.

Repository Creation and Configuration

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

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

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

In Subversion 1.2, a repository is created with an FSFS back-end by default (see the section called "Repository Data Stores"). The back-end can be explicitly chosen with the `--fs-type` argument:

```
$ svnadmin create --fs-type fsfs /path/to/repos
$ svnadmin create --fs-type bdb /path/to/other/repos
```

Warning

Do not create a Berkeley DB repository on a network share—it *cannot* exist on a remote filesystem such as NFS, AFS, or Windows SMB. Berkeley DB requires that the underlying filesystem implement strict POSIX locking semantics, and more importantly, the ability to map files directly into process memory. Almost no network filesystems provide these features. If you attempt to use Berkeley DB on a network share, the results are unpredictable—you may see mysterious errors right away, or it may be months before you discover that your repository database is subtly corrupted.

If you need multiple computers to access the repository, you create an FSFS repository on the network share, not a Berkeley DB repository. Or better yet, set up a real server process (such as Apache or **svnserve**), store the repository on a local filesystem which the server can access, and make the repository available over a network. Chapter 6, *Server Configuration* covers this process in detail.

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

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

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

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

conf
A directory containing repository configuration files.

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

db
Where all of your versioned data resides. This directory is either a Berkeley DB environment (full of DB tables

and other things), or is an FSFS environment containing revision files.

format

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

hooks

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

locks

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

README.txt

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

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

Hook Scripts

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

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

but doesn't work when run by Subversion. Be sure to explicitly set environment variables in your hook and/or use absolute paths to programs.

There are nine hooks implemented by the Subversion repository:

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

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

The Subversion distribution includes some access control scripts (located in the `tools/hook-scripts` directory of the Subversion source tree) that can be called from **pre-commit** to implement fine-grained write-access control. Another option is to use the **mod_authz_svn** Apache httpd module, which provides both read and write access control on individual directories (see the section called "Per-Directory Access Control"). In a future version of Subversion, we plan to implement access control lists (ACLs) directly in the filesystem.

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

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

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

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

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

`post-revprop-change`
As mentioned earlier, this hook is the counterpart of the `pre-revprop-change` hook. In fact, for the sake of paranoia this script will not run unless the `pre-revprop-change` hook exists. When both of these hooks are

present, the `post-revprop-change` hook runs just after a revision property has been changed, and is typically used to send an email containing the new value of the changed property. The repository passes four arguments to this hook: the path to the repository, the revision on which the property exists, the authenticated username of the person making the change, and the name of the property itself.

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

`pre-lock`

This hook runs whenever someone attempts to lock a file. It can be used to prevent locks altogether, or to create a more complex policy specifying exactly which users are allowed to lock particular paths. If the hook notices a pre-existing lock, then it can also decide whether a user is allowed to “steal” the existing lock. The repository passes three arguments to the hook: the path to the repository, the path being locked, and the user attempting to perform the lock. If the program returns a non-zero exit value, the lock action is aborted and anything printed to stderr is marshalled back to the client.

`post-lock`

This hook runs after a path is locked. The locked path is passed to the hook's stdin, and the hook also receives two arguments: the path to the repository, and the user who performed the lock. The hook is then free to send email notification or record the event in any way it chooses. Because the lock already happened, the output of the hook is ignored.

`pre-unlock`

This hook runs whenever someone attempts to remove a lock on a file. It can be used to create policies that specify which users are allowed to unlock particular paths. It's particularly important for determining policies about lock breakage. If user A locks a file, is user B allowed to break the lock? What if the lock is more than a week old? These sorts of things can be decided and enforced by the hook. The repository passes three arguments to the hook: the path to the repository, the path being unlocked, and the user attempting to remove the lock. If the program returns a non-zero exit value, the unlock action is aborted and anything printed to stderr is marshalled back to the client.

`post-unlock`

This hook runs after a path is unlocked. The unlocked path is passed to the hook's stdin, and the hook also receives two arguments: the path to the repository, and the user who removed the lock. The hook is then free to send email notification or record the event in any way it chooses. Because the lock removal already happened, the output of the hook is ignored.

Warning

Do not attempt to modify the transaction using hook scripts. A common example of this would be to automatically set properties such as `svn:eol-style` or `svn:mime-type` during the commit. While this might seem like a good idea, it causes problems. The main problem is that the client does not know about the change made by the hook script, and there is no way to inform the client that it is out-of-date. This inconsistency can lead to surprising and unexpected behavior.

Instead of attempting to modify the transaction, it is much better to *check* the transaction in the `pre-commit` hook and reject the commit if it does not meet the desired requirements.

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

Berkeley DB Configuration

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

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

The Berkeley configuration file for your repository is located in the `db` environment directory, at `repos/db/DB_CONFIG`. Subversion itself creates this file when it creates the rest of the repository. The file initially contains some default options, as well as pointers to the Berkeley DB online documentation so you can read about what those options do. Of course, you are free to add any of the supported Berkeley DB options to your `DB_CONFIG` file. Just be aware that while Subversion never attempts to read or interpret the contents of the file, and makes no use of the option settings in it, you'll want to avoid any configuration changes that may cause Berkeley DB to behave in a fashion that is unexpected by the rest of the Subversion code. Also, changes made to `DB_CONFIG` won't take effect until you recover the database environment (using **svnadmin recover**).

Repository Maintenance

Maintaining a Subversion repository can be a daunting task, mostly due to the complexities inherent in systems which have a database backend. Doing the task well is all about knowing the tools—what they are, when to use them, and how to use them. This section will introduce you to the repository administration tools provided by Subversion, and how to wield them to accomplish tasks such as repository migrations, upgrades, backups and cleanups.

An Administrator's Toolkit

Subversion provides a handful of utilities useful for creating, inspecting, modifying and repairing your repository. Let's look more closely at each of those tools. Afterward, we'll briefly examine some of the utilities included in the Berkeley DB distribution that provide functionality specific to your repository's database backend not otherwise provided by Subversion's own tools.

svnlook

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

svnlook has a straightforward syntax:

```
$ svnlook help
general usage: svnlook SUBCOMMAND REPOS_PATH [ARGS & OPTIONS ...]
```

```
Note: any subcommand which takes the '--revision' and '--transaction'
      options will, if invoked without one of those options, act on
      the repository's youngest revision.
Type "svnlook help <subcommand>" for help on a specific subcommand.
…
```

Nearly every one of **svnlook**'s subcommands can operate on either a revision or a transaction tree, printing information about the tree itself, or how it differs from the previous revision of the repository. You use the `--revision` and `--transaction` options to specify which revision or transaction, respectively, to examine. Note that while revision numbers appear as natural numbers, transaction names are alphanumeric strings. Keep in mind that the filesystem only allows browsing of uncommitted transactions (transactions that have not resulted in a new revision). Most repositories will have no such transactions, because transactions are usually either committed (which disqualifies them from viewing) or aborted and removed.

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

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

The only exception to these rules about subcommands is the **svnlook youngest** subcommand, which takes no options, and simply prints out the allows scripts and other wrappers around this command to make intelligent decisions about the log message, such as how much memory to allocate for the message, or at least how many bytes to skip in the event that this output is not the last bit of data in the stream.

Another common use of **svnlook** is to actually view the contents of a revision or transaction tree. The **svnlook tree** command displays the directories and files in the requested tree. If you supply the `--show-ids` option, it will also show the filesystem node revision IDs for each of those paths (which is generally of more use to developers than to users).

```
$ svnlook tree /path/to/repos --show-ids
/ <0.0.1>
 A/ <2.0.1>
  B/ <4.0.1>
   lambda <5.0.1>
   E/ <6.0.1>
    alpha <7.0.1>
    beta <8.0.1>
   F/ <9.0.1>
  mu <3.0.1>
  C/ <a.0.1>
  D/ <b.0.1>
   gamma <c.0.1>
   G/ <d.0.1>
    pi <e.0.1>
    rho <f.0.1>
    tau <g.0.1>
   H/ <h.0.1>
    chi <i.0.1>
    omega <k.0.1>
    psi <j.0.1>
 iota <1.0.1>
```

Once you've seen the layout of directories and files in your tree, you can use commands like **svnlook cat**, **svnlook propget**, and **svnlook proplist** to dig into the details of those files and directories.

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

`author`
Print the tree's author.

`cat`
Print the contents of a file in the tree.

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

`date`
Print the tree's datestamp.

`diff`
Print unified diffs of changed files.

`dirs-changed`
List the directories in the tree that were themselves changed, or whose file children were changed.

`history`

Display interesting points in the history of a versioned path (places where modifications or copies occurred).

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

`lock`
If a path is locked, describe the lock attributes.

`log`
Print the tree's log message.

`propget`
Print the value of a property on a path in the tree.

`proplist`
Print the names and values of properties set on paths in the tree.

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

`uuid`
Print the repository's UUID— *U*niversal *U*nique *ID*entifier.

`youngest`
Print the youngest revision number.

svnadmin

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

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

Available subcommands:
   create
   deltify
   dump
   help (?, h)
…
```

We've already mentioned **svnadmin**'s `create` subcommand (see the section called “Repository Cre-

`hotcopy`
Make a hot copy of a repository. You can run this command at any time and make a safe copy of the repository, regardless if other processes are using the repository.

`list-dblogs`
(Berkeley DB repositories only.) List the paths of Berkeley DB log files associated with the repository. This list includes all log files—those still in use by Subversion, as well as those no longer in use.

`list-unused-dblogs`
(Berkeley DB repositories only.) List the paths of Berkeley DB log files associated with, but no longer used by, the repository. You may safely remove these log files from the repository layout, possibly archiving them for use in the event that you ever need to perform a catastrophic recovery of the repository.

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

`lslocks`
List and describe any locks that exist in the repository.

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

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

`rmlocks`
Unconditionally remove locks from listed paths.

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

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

`verify`
Verify the contents of the repository. This includes, among other things, checksum comparisons of the versioned data stored in the repository.

svndumpfilter

Since Subversion stores everything in an opaque database system, attempting manual tweaks is unwise, if not quite difficult. And once data has been stored in your repository, Subversion generally doesn't provide an easy way to remove that data. [3] But inevitably, there will be times when you would like to manipulate the history of your repository. You might need to strip out all instances of a file that was accidentally added to the repository (and shouldn't be there for whatever reason). Or, perhaps you have multiple projects sharing a single repository, and you decide to split them up into their own repositories. To accomplish tasks like this, administrators need a more manageable and malleable representation of the data in their repositories—the Subversion repository dump format.

The Subversion repository dump format is a human-readable representation of the changes that you've made to your versioned data over time. You use the **svnadmin dump** command to generate the dump

[3]That, by the way, is a *feature*, not a bug.

data, and **svnadmin load** to populate a new repository with it (see the section called “Migrating a Repository”). The great thing about the human-readability aspect of the dump format is that, if you aren't careless about it, you can manually inspect and modify it. Of course, the downside is that if you have two years' worth of repository activity encapsulated in what is likely to be a very large dump file, it could take you a long, long time to manually inspect and modify it.

While it won't be the most commonly used tool at the administrator's disposal, **svndumpfilter** provides a very particular brand of useful functionality—the ability to quickly and easily modify that dump data by acting as a path-based filter. Simply give it either a list of paths you wish to keep, or a list of paths you wish to not keep, then pipe your repository dump data through this filter. The result will be a modified stream of dump data that contains only the versioned paths you (explicitly or implicitly) requested.

The syntax of **svndumpfilter** is as follows:

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

Available subcommands:
   exclude
   include
   help (?, h)
```

There are only two interesting subcommands. They allow you to make the choice between explicit or implicit inclusion of paths in the stream:

`exclude`
Filter out a set of paths from the dump data stream.

`include`
Allow only the requested set of paths to pass through the dump data stream.

Let's look a realistic example of how you might use this program. We discuss elsewhere (see the section called “Choosing a Repository Layout”) the process of deciding how to choose a layout for the data in your repositories—using one repository per project or combining them, arranging stuff within your repository, and so on. But sometimes after new revisions start flying in, you rethink your layout and would like to make some changes. A common change is the decision to move multiple projects which are sharing a single repository into separate repositories for each project.

Our imaginary repository contains three projects: `calc`, `calendar`, and `spreadsheet`. They have been living side-by-side in a layout like this:

```
/
   calc/
      trunk/
      branches/
      tags/
   calendar/
      trunk/
      branches/
      tags/
   spreadsheet/
      trunk/
      branches/
      tags/
```

To get these three projects into their own repositories, we first dump the whole repository:

```
$ svnadmin dump /path/to/repos > repos-dumpfile
* Dumped revision 0.
* Dumped revision 1.
* Dumped revision 2.
* Dumped revision 3.
…
$
```

Next, run that dump file through the filter, each time including only one of our top-level directories, and resulting in three new dump files:

```
$ cat repos-dumpfile | svndumpfilter include calc > calc-dumpfile
…
$ cat repos-dumpfile | svndumpfilter include calendar > cal-dumpfile
…
$ cat repos-dumpfile | svndumpfilter include spreadsheet > ss-dumpfile
…
$
```

At this point, you have to make a decision. Each of your dump files will create a valid repository, but will preserve the paths exactly as they were in the original repository. This means that even though you would have a repository solely for your `calc` project, that repository would still have a top-level directory named `calc`. If you want your `trunk`, `tags`, and `branches` directories to live in the root of your repository, you might wish to edit your dump files, tweaking the `Node-path` and `Node-copyfrom-path` headers to no longer have that first `calc/` path component. Also, you'll want to remove the section of dump data that creates the `calc` directory. It will look something like:

```
Node-path: calc
Node-action: add
Node-kind: dir
Content-length: 0
```

Warning

If you do plan on manually editing the dump file to remove a top-level directory, make sure that your editor is not set to automatically convert end-lines to the native format (e.g. \r\n to \n) as the content will then not agree with the metadata and this will render the dump file useless.

All that remains now is to create your three new repositories, and load each dump file into the right repository:

```
$ svnadmin create calc; svnadmin load calc < calc-dumpfile
<<< Started new transaction, based on original revision 1
     * adding path : Makefile ... done.
     * adding path : button.c ... done.
…
$ svnadmin create calendar; svnadmin load calendar < cal-dumpfile
<<< Started new transaction, based on original revision 1
     * adding path : Makefile ... done.
```

```
     * adding path : cal.c ... done.
…
$ svnadmin create spreadsheet; svnadmin load spreadsheet < ss-dumpfile
<<< Started new transaction, based on original revision 1
     * adding path : Makefile ... done.
     * adding path : ss.c ... done.
…
$
```

Both of **svndumpfilter**'s subcommands accept options for deciding how to deal with “empty” revisions. If a given revision contained only changes to paths that were filtered out, that now-empty revision could be considered uninteresting or even unwanted. So to give the user control over what to do with those revisions, **svndumpfilter** provides the following command-line options:

`--drop-empty-revs`
Do not generate empty revisions at all—just omit them.

`--renumber-revs`
If empty revisions are dropped (using the `--drop-empty-revs` option), change the revision numbers of the remaining revisions so that there are no gaps in the numeric sequence.

`--preserve-revprops`
If empty revisions are not dropped, preserve the revision properties (log message, author, date, custom properties, etc.) for those empty revisions. Otherwise, empty revisions will only contain the original datestamp, and a generated log message that indicates that this revision was emptied by **svndumpfilter**.

While **svndumpfilter** can be very useful, and a huge timesaver, there are unfortunately a couple of gotchas. First, this utility is overly sensitive to path semantics. Pay attention to whether paths in your dump file are specified with or without leading slashes. You'll want to look at the `Node-path` and `Node-copyfrom-path` headers.

```
…
Node-path: spreadsheet/Makefile
…
```

If the paths have leading slashes, you should include leading slashes in the paths you pass to **svndumpfilter include** and **svndumpfilter exclude** (and if they don't, you shouldn't). Further, if your dump file has an inconsistent usage of leading slashes for some reason, [4] you should probably normalize those paths so they all have, or lack, leading slashes.

Also, copied paths can give you some trouble. Subversion supports copy operations in the repository, where a new path is created by copying some already existing path. It is possible that at some point in

[4]While **svnadmin dump** has a consistent leading slash policy—to not include them—other programs which generate dump data might not be so consistent.

If you're using a Berkeley DB repository, then all of your versioned filesystem's structure and data live in a set of database tables within the `db` subdirectory of your repository. This subdirectory is a regular Berkeley DB environment directory, and can therefore be used in conjunction with any of the Berkeley database tools (you can see the documentation for these tools at Sleepycat's website, http://www.sleepycat.com/).

For day-to-day Subversion use, these tools are unnecessary. Most of the functionality typically needed for Subversion repositories has been duplicated in the **svnadmin** tool. For example, **svnadmin list-unused-dblogs** and **svnadmin list-dblogs** perform a subset of what is provided by the Berkeley **db_archive** command, and **svnadmin recover** reflects the common use cases of the **db_recover** utility.

There are still a few Berkeley DB utilities that you might find useful. The **db_dump** and **db_load** programs write and read, respectively, a custom file format which describes the keys and values in a Berkeley DB database. Since Berkeley databases are not portable across machine architectures, this format is a useful way to transfer those databases from machine to machine, irrespective of architecture or operating system. Also, the **db_stat** utility can provide useful information about the status of your Berkeley DB environment, including detailed statistics about the locking and storage subsystems.

Repository Cleanup

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

- modifying commit log messages,
- removing dead transactions,
- recovering “wedged” repositories, and
- migrating repository contents to a different repository.

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

Sometimes a user will have an error in her log message (a misspelling or some misinformation, perhaps). If the repository is configured (using the `pre-revprop-change` and `post-revprop-change` hooks; see the section called “Hook Scripts”) to accept changes to this log message after the commit is finished, then the user can “fix” her log message remotely using the **svn** program's `propset` command (see Chapter 9, *Subversion Complete Reference*). However, because of the potential to lose information forever, Subversion repositories are not, by default, configured to allow changes to unversioned properties—except by an administrator.

triggered, and therefore must be setup to accept changes of this nature. But an administrator can get around these protections by passing the `--bypass-hooks` option to the **svnadmin setlog** command.

Warning

Remember, though, that by bypassing the hooks, you are likely avoiding such things as email notifications of property changes, backup systems which track unversioned property changes, and so on. In other words, be very careful about what you are changing, and how you change it.

Another common use of **svnadmin** is to query the repository for outstanding—possibly dead—Subversion transactions. In the event that a commit should fail, the transaction is usually cleaned up. That is, the transaction itself is removed from the repository, and any data associated with (and only with) that transaction is removed as well. Occasionally, though, a failure occurs in such a way that the cleanup of the transaction never happens. This could happen for several reasons: perhaps the client operation was inelegantly terminated by the user, or a network failure might have occurred in the middle of an operation, etc. Regardless of the reason, dead transactions can happen. They don't do any real harm, other than consuming a small bit of disk space. A fastidious administrator may nonetheless want to remove them.

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

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

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

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

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

Example 5.1. txn-info.sh (Reporting Outstanding Transactions)

```
#!/bin/sh

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

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

```
fi

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

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

```
$ txn-info.sh myrepos
---[ Transaction 19 ]-------------------------------------------
sally
2001-09-04 11:57:19 -0500 (Tue, 04 Sep 2001)
0
---[ Transaction 3a1 ]-------------------------------------------
harry
2001-09-10 16:50:30 -0500 (Mon, 10 Sep 2001)
39
Trying to commit over a faulty network.
---[ Transaction a45 ]-------------------------------------------
sally
2001-09-12 11:09:28 -0500 (Wed, 12 Sep 2001)
0
$
```

A long-abandoned transaction usually represents some sort of failed or interrupted commit. A transaction's datestamp can provide interesting information—for example, how likely is it that an operation begun nine months ago is still active?

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

Managing Disk Space

While the cost of storage has dropped incredibly in the past few years, disk usage is still a valid concern for administrators seeking to version large amounts of data. Every additional byte consumed by the live repository is a byte that needs to be backed up offsite, perhaps multiple times as part of rotating backup schedules. If using a Berkeley DB repository, the primary storage mechanism is a complex database system, it is useful to know what pieces of data need to remain on the live site, which need to be backed up, and which can be safely removed. This section is specific to Berkeley DB; FSFS repositories have no extra data to be cleaned up or reclaimed.

Until recently, the largest offender of disk space usage with respect to Subversion repositories was the log files to which Berkeley DB performs its pre-writes before modifying the actual database files. These files capture all the actions taken along the route of changing the database from one state to another—while the database files reflect at any given time some state, the log files contain all the many changes along the way between states. As such, they can start to accumulate quite rapidly.

Fortunately, beginning with the 4.2 release of Berkeley DB, the database environment has the ability to remove its own unused log files without any external procedures. Any repositories created using an **svnadmin** which is compiled against Berkeley DB version 4.2 or greater will be configured for this automatic log file removal. If you don't want this feature enabled, simply pass the `--bdb-log-keep` option to the **svnadmin create** command. If you forget to do this, or change your mind at a later time, simple edit the `DB_CONFIG` file found in your repository's `db` directory, comment out the line which contains the `set_flags DB_LOG_AUTOREMOVE` directive, and then run **svnadmin recover** on your repository to force the configuration changes to take effect. See the section called “Berkeley DB Configuration” for more information about database configuration.

Without some sort of automatic log file removal in place, log files will accumulate as you use your repository. This is actually somewhat of a feature of the database system—you should be able to recreate your entire database using nothing but the log files, so these files can be useful for catastrophic database recovery. But typically, you'll want to archive the log files that are no longer in use by Berkeley DB, and then remove them from disk to conserve space. Use the **svnadmin list-unused-dblogs** command to list the unused log files:

```
$ svnadmin list-unused-dblogs /path/to/repos
/path/to/repos/log.0000000031
/path/to/repos/log.0000000032
/path/to/repos/log.0000000033

$ svnadmin list-unused-dblogs /path/to/repos | xargs rm
## disk space reclaimed!
```

To keep the size of the repository as small as possible, Subversion uses *deltification* (or, “deltified storage”) within the repository itself. Deltification involves encoding the representation of a chunk of data as a collection of differences against some other chunk of data. If the two pieces of data are very similar, this deltification results in storage savings for the deltified chunk—rather than taking up space equal to the size of the original data, it only takes up enough space to say, “I look just like this other piece of data over here, except for the following couple of changes”. Specifically, each time a new version of a file is information.)

[5]E.g.: hard drive + huge electromagnet = disaster.

The most common reason to dump and load a Subversion repository is due to changes in Subversion itself. As Subversion matures, there are times when certain changes made to the back-end database schema cause Subversion to be incompatible with previous versions of the repository. Other reasons for dumping and loading might be to migrate a Berkeley DB repository to a new OS or CPU architecture, or to switch between Berkeley DB and FSFS back-ends. The recommended course of action is relatively simple:

1. Using your *current* version of **svnadmin**, dump your repositories to dump files.

2. Upgrade to the new version of Subversion.

3. Move your old repositories out of the way, and create new empty ones in their place using your *new* **svnadmin**.

4. Again using your *new* **svnadmin**, load your dump files into their respective, just-created repositories.

5. Be sure to copy any customizations from your old repositories to the new ones, including `DB_CONFIG` files and hook scripts. You'll want to pay attention to the release notes for the new release of Subversion to see if any changes since your last upgrade affect those hooks or configuration options.

6. If the migration process made your repository accessible at a different URL (e.g. moved to a different computer, or is being accessed via a different schema), then you'll probably want to tell your users to run **svn switch --relocate** on their existing working copies. See svn switch.

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

```
$ svnlook youngest myrepos
26
$ svnadmin dump myrepos > dumpfile
* Dumped revision 0.
* Dumped revision 1.
* Dumped revision 2.
…
* Dumped revision 25.
* Dumped revision 26.
```

At the end of the process, you will have a single file (`dumpfile` in the previous example) that contains all the data stored in your repository in the requested range of revisions. Note that **svnadmin dump** is reading revision trees from the repository just like any other "reader" process would (**svn checkout**, for example). So it's safe to run this command at any time.

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

```
$ svnadmin load newrepos < dumpfile
```

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

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

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

…

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

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

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

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

The result of a load is new revisions added to a repository—the same thing you get by making commits against that repository from a regular Subversion client. And just as in a commit, you can use hook scripts to perform actions before and after each of the commits made during a load process. By passing the `--use-pre-commit-hook` and `--use-post-commit-hook` options to **svnadmin load**, you can instruct Subversion to execute the pre-commit and post-commit hook scripts, respectively, for each loaded revision. You might use these, for example, to ensure that loaded revisions pass through the same validation steps that regular commits pass through. Of course, you should use these options with care—if your post-commit hook sends emails to a mailing list for each new commit, you might not want to spew hundreds or thousands of commit emails in rapid succession at that list for each of the loaded revisions! You can read more about the use of hook scripts in the section called "Hook Scripts".

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

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

By default, the dump file will be quite large—much larger than the repository itself. That's because every version of every file is expressed as a full text in the dump file. This is the fastest and simplest behavior, and nice if you're piping the dump data directly into some other process (such as a compression program, filtering program, or into a loading process). But if you're creating a dump file for longer-term storage, you'll likely want to save disk space by using the `--deltas` switch. With this option, successive revisions of files will be output as compressed, binary differences—just as file revisions are stored in a repository. This option is slower, but results in a dump file much closer in size to the original repository.

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

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

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

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

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

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

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

```
$ svnadmin load newrepos < dumpfile1
$ svnadmin load newrepos < dumpfile2
$ svnadmin load newrepos < dumpfile3
```

Another neat trick you can perform with this `--incremental` option involves appending to an existing dump file a new range of dumped revisions. For example, you might have a `post-commit` hook that simply appends the repository dump of the single revision that triggered the hook. Or you might have a script that runs nightly to append dump file data for all the revisions that were added to the repository since the last time the script ran. Used like this, **svnadmin**'s `dump` and `load` commands can be a valuable means by which to backup changes to your repository over time in case of a system crash or some other catastrophic event.

The dump format can also be used to merge the contents of several different repositories into a single repository. By using the `--parent-dir` option of **svnadmin load**, you can specify a new virtual root directory for the load process. That means if you have dump files for three repositories, say `calc-dumpfile`, `cal-dumpfile`, and `ss-dumpfile`, you can first create a new repository to hold them all:

```
$ svnadmin create /path/to/projects
$
```

Then, make new directories in the repository which will encapsulate the contents of each of the three previous repositories:

```
$ svn mkdir -m "Initial project roots" \
      file:///path/to/projects/calc \
      file:///path/to/projects/calendar \
      file:///path/to/projects/spreadsheet
```

```
Committed revision 1.
$
```

Lastly, load the individual dump files into their respective locations in the new repository:

```
$ svnadmin load /path/to/projects --parent-dir calc < calc-dumpfile
…
$ svnadmin load /path/to/projects --parent-dir calendar < cal-dumpfile
…
$ svnadmin load /path/to/projects --parent-dir spreadsheet < ss-dumpfile
…
$
```

We'll mention one final way to use the Subversion repository dump format—conversion from a different storage mechanism or version control system altogether. Because the dump file format is, for the most part, human-readable, [6] it should be relatively easy to describe generic sets of changes—each of which should be treated as a new revision—using this file format. In fact, the **cvs2svn** utility (see the section called “Converting a Repository from CVS to Subversion”) uses the dump format to represent the contents of a CVS repository so that those contents can be copied into a Subversion repository.

Repository Backup

Despite numerous advances in technology since the birth of the modern computer, one thing unfortunately rings true with crystalline clarity—sometimes, things go very, very awry. Power outages, network connectivity dropouts, corrupt RAM and crashed hard drives are but a taste of the evil that Fate is poised to unleash on even the most conscientious administrator. And so we arrive at a very important topic—how to make backup copies of your repository data.

There are generally two types of backup methods available for Subversion repository administrators—incremental and full. We discussed in an earlier section of this chapter how to use **svnadmin dump --incremental** to perform an incremental backup (see the section called “Migrating a Repository”). Essentially, the idea is to only backup at a given time the changes to the repository since the last time you made a backup.

A full backup of the repository is quite literally a duplication of the entire repository directory (which includes either Berkeley database or FSFS environment). Now, unless you temporarily disable all other access to your repository, simply doing a recursive directory copy runs the risk of generating a faulty backup, since someone might be currently writing to the database.

In the case of Berkeley DB, Sleepycat documents describe a certain order in which database files can be copied that will guarantee a valid backup copy. And a similar ordering exists for FSFS data. Better still, you don't have to implement these algorithms yourself, because the Subversion development team has already done so. The **hot-backup.py** script is found in the `tools/backup/` directory of the Subversion source distribution. Given a repository path and a backup location, **hot-backup.py**—which is really just a more intelligent wrapper around the **svnadmin hotcopy** command—will perform the necessary steps for backing up your live repository—without requiring that you bar public repository access at all—and then will clean out the dead Berkeley log files from your live repository.

Even if you also have an incremental backup, you might want to run this program on a regular basis. For example, you might consider adding **hot-backup.py** to a program scheduler (such as **cron** on Unix systems). Or, if you prefer fine-grained backup solutions, you could have your post-commit hook script call **hot-backup.py** (see the section called “Hook Scripts”), which will then cause a new backup of your repository to occur with every new revision created. Simply add the following to the `hooks/post-commit` script in your live repository directory:

[6] The Subversion repository dump format resembles an RFC-822 format, the same type of format used for most email.

```
(cd /path/to/hook/scripts; ./hot-backup.py ${REPOS} /path/to/backups &)
```

The resulting backup is a fully functional Subversion repository, able to be dropped in as a replacement for your live repository should something go horribly wrong.

There are benefits to both types of backup methods. The easiest is by far the full backup, which will always result in a perfect working replica of your repository. This again means that should something bad happen to your live repository, you can restore from the backup with a simple recursive directory copy. Unfortunately, if you are maintaining multiple backups of your repository, these full copies will each eat up just as much disk space as your live repository.

Incremental backups using the repository dump format are excellent to have on hand if the database schema changes between successive versions of Subversion itself. Since a complete repository dump and load are generally required to upgrade your repository to the new schema, it's very convenient to already have half of that process (the dump part) finished. Unfortunately, the creation of—and restoration from—incremental backups takes longer, as each commit is effectively replayed into either the dump file or the repository.

In either backup scenario, repository administrators need to be aware of how modifications to unversioned revision properties affect their backups. Since these changes do not themselves generate new revisions, they will not trigger post-commit hooks, and may not even trigger the pre-revprop-change and post-revprop-change hooks. [7] And since you can change revision properties without respect to chronological order—you can change any revision's properties at any time—an incremental backup of the latest few revisions might not catch a property modification to a revision that was included as part of a previous backup.

Generally speaking, only the truly paranoid would need to backup their entire repository, say, every time a commit occurred. However, assuming that a given repository has some other redundancy mechanism in place with relatively fine granularity (like per-commit emails), a hot backup of the database might be something that a repository administrator would want to include as part of a system-wide nightly backup. For most repositories, archived commit emails alone provide sufficient redundancy as restoration sources, at least for the most recent few commits. But it's your data—protect it as much as you'd like.

Often, the best approach to repository backups is a diversified one. You can leverage combinations of full and incremental backups, plus archives of commit emails. The Subversion developers, for example, back up the Subversion source code repository after every new revision is created, and keep an archive of all the commit and property change notification emails. Your solution might be similar, but should be catered to your needs and that delicate balance of convenience with paranoia. And while all of this might not save your hardware from the iron fist of Fate, [8] it should certainly help you recover from those trying times.

Adding Projects

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

[7] **svnadmin setlog** can be called in a way that bypasses the hook interface altogether.
[8] You know—the collective term for all of her “fickle fingers”.

Choosing a Repository Layout

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

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

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

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

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

After deciding how to organize your projects with respect to repositories, you'll probably want to think about directory hierarchies in the repositories themselves. Because Subversion uses regular directory copies for branching and tagging (see Chapter 4, *Branching and Merging*), the Subversion community recommends that you choose a repository location for each *project root*—the “top-most” directory which contains data related to that project—and then create three subdirectories beneath that root: `trunk`, meaning the directory under which the main project development occurs; `branches`

[9]The `trunk`, `tags`, and `branches` trio are sometimes referred to as “the TTB directories”.

Note that it doesn't matter where in your repository each project root is. If you have only one project per

```
…
Committed revision 1.
$ cd ..
$ rm -rf tmpdir
$
```

You can verify the results of the import by running the **svn list** command:

```
$ svn list --verbose file:///path/to/repos
      1 harry               May 08 21:48 projectA/
      1 harry               May 08 21:48 projectB/
…
$
```

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

Summary

By now you should have a basic understanding of how to create, configure, and maintain Subversion repositories. We've introduced you to the various tools that will assist you with this task. Throughout the chapter, we've noted common administration pitfalls, and suggestions for avoiding them.

All that remains is for you to decide what exciting data to store in your repository, and finally, how to make it available over a network. The next chapter is all about networking.

Chapter 6. Server Configuration

A Subversion repository can be accessed simultaneously by clients running on the same machine on which the repository resides using the `file:///` method. But the typical Subversion setup involves a single server machine being accessed from clients on computers all over the office—or, perhaps, all over the world.

This section describes how to get your Subversion repository exposed outside its host machine for use by remote clients. We will cover Subversion's currently available server mechanisms, discussing the configuration and use of each. After reading this section, you should be able to decide which networking setup is right for your needs, and understand how to enable such a setup on your host computer.

Overview

Subversion was designed with an abstract network layer. This means that a repository can be programmatically accessed by any sort of server process, and the client “repository access” API allows programmers to write plugins that speak relevant network protocols. In theory, Subversion can use an infinite number of network implementations. In practice, there are only two servers at the time of writing.

Apache is an extremely popular webserver; using the **mod_dav_svn** module, Apache can access a repository and make it available to clients via the WebDAV/DeltaV protocol, which is an extension of HTTP. In the other corner is **svnserve**: a small, standalone server program that speaks a custom protocol with clients. Table 6-1 presents a comparison of the two servers.

Note that Subversion, as an open-source project, does not officially endorse any server as “primary” or “official”. Neither network implementation is treated as a second-class citizen; each server has advantages and disadvantages. In fact, it's possible for different servers to run in parallel, each accessing your repositories in its own way, and each without hindering the other (see the section called “Supporting Multiple Repository Access Methods”). Table 6.1, “Network Server Comparison” gives a brief overview and comparison of the two available Subversion servers—as an administrator, it's up to you to choose whatever works best for you and your users.

Table 6.1. Network Server Comparison

Feature	Apache + mod_dav_svn	svnserve
Authentication options		

Network Model

This section is a general discussion of how a Subversion client and server interact with one another, regardless of the network implementation you're using. After reading, you'll have a good understanding of how a server can behave and the different ways in which a client can be configured to respond.

Requests and Responses

The Subversion client spends most of its time managing working copies. When it needs information from a repository, however, it makes a network request, and the server responds with an appropriate answer. The details of the network protocol are hidden from the user; the client attempts to access a URL, and depending on the URL schema, a particular protocol is used to contact the server (see Repository URLs). Users can run **svn --version** to see which URL schemas and protocols the client knows how to use.

When the server process receives a client request, it typically demands that the client identify itself. It issues an authentication challenge to the client, and the client responds by providing *credentials* back to the server. Once authentication is complete, the server responds with the original information the client asked for. Notice that this system is different from systems like CVS, where the client pre-emptively offers credentials ("logs in") to the server before ever making a request. In Subversion, the server "pulls" credentials by challenging the client at the appropriate moment, rather than the client "pushing" them. This makes certain operations more elegant. For example, if a server is configured to allow anyone in the world to read a repository, then the server will never issue an authentication challenge when a client attempts to **svn checkout**.

If the client's network request writes new data to the repository (e.g. **svn commit**), then a new revision tree is created. If the client's request was authenticated, then the authenticated user's name is stored as the value of the `svn:author` property on the new revision (see the section called "Unversioned Properties"). If the client was not authenticated (in other words, the server never issued an authentication challenge), then the revision's `svn:author` property is empty. [1]

Client Credentials Caching

Many servers are configured to require authentication on every request. This can become a big annoyance to users, who are forced to type their passwords over and over again.

Happily, the Subversion client has a remedy for this: a built-in system for caching authentication credentials on disk. By default, whenever the command-line client successfully responds to a server's authentication challenge, it saves the credentials in the user's private runtime configuration area—in `~/.subversion/auth/` on Unix-like systems or `%APPDATA%/Subversion/auth/` on Windows. (The runtime area is covered in more detail in the section called "Runtime Configuration Area".) Successful credentials are cached on disk, keyed on a combination of hostname, port, and authentication realm.

When the client receives an authentication challenge, it first looks for the appropriate credentials in the user's disk cache; if not present, or if the cached credentials fail to authenticate, then the client simply prompts the user for the information.

Security-conscious people may be thinking to themselves, "Caching passwords on disk? That's terrible! You should never do that!" Please remain calm, it's not as dangerous as it sounds.

- The `auth/` caching area is permission-protected so that only the user (owner) can read data from it, not the world at large. The operating system's own file permissions are protecting the password.

[1] This problem is actually a FAQ, resulting from a misconfigured server setup.

- On Windows 2000 and later, the Subversion client uses standard Windows cryptography services to encrypt the password on disk. Because the encryption key is managed by Windows and is tied to the user's own login credentials, only the user can decrypt the cached password. (Note: if the user's Windows account password is reset by an administrator, all of the cached passwords become undecipherable. The Subversion client will behave as if they don't exist, prompting for passwords when required.)

- For the truly paranoid willing to sacrifice all convenience, it's possible to disable credential caching altogether.

To disable caching for a single command, pass the `--no-auth-cache` option:

```
$ svn commit -F log_msg.txt --no-auth-cache
Authentication realm: <svn://host.example.com:3690> example realm
Username:  joe
Password for 'joe':

Adding         newfile
Transmitting file data .
Committed revision 2324.

# password was not cached, so a second commit still prompts us

$ svn delete newfile
$ svn commit -F new_msg.txt
Authentication realm: <svn://host.example.com:3690> example realm
Username:  joe
…
```

Or, if you want to disable credential caching permanently, you can edit your runtime `config` file (located next to the `auth/` directory). Simply set `store-auth-creds` to `no`, and no credentials will be cached on disk, ever.

```
[auth]
store-auth-creds = no
```

Sometimes users will want to remove specific credentials from the disk cache. To do this, you need to navigate into the `auth/` area and manually delete the appropriate cache file. Credentials are cached in individual files; if you look inside each file, you will see keys and values. The `svn:realmstring` key describes the particular server realm that the file is associated with:

```
$ ls ~/.subversion/auth/svn.simple/
5671adf2865e267db74f09ba6f872c28
3893ed123b39500bca8a0b382839198e
5c3c22968347b390f349ff340196ed39

$ cat ~/.subversion/auth/svn.simple/5671adf2865e267db74f09ba6f872c28

K 8
username
V 3
joe
K 8
password
V 4
blah
```

```
K 15
svn:realmstring
V 45
<https://svn.domain.com:443> Joe's repository
END
```

Once you have located the proper cache file, just delete it.

One last word about client authentication behavior: a bit of explanation about the `--username` and `--password` options is needed. Many client subcommands accept these options; however it is important to understand using these options *does not* automatically send credentials to the server. As discussed earlier, the server "pulls" credentials from the client when it deems necessary; the client cannot "push" them at will. If a username and/or password are passed as options, they will *only* be presented to the server if the server requests them. [2] Typically, these options are used when:

- the user wants to authenticate as a different user than her system login name, or
- a script wants to authenticate without using cached credentials.

Here is a final summary that describes how a Subversion client behaves when it receives an authentication challenge:

1. Check whether the user specified any credentials as command-line options, via `--username` and/or `--password`. If not, or if these options fail to authenticate successfully, then
2. Look up the server's realm in the runtime `auth/` area, to see if the user already has the appropriate credentials cached. If not, or if the cached credentials fail to authenticate, then
3. Resort to prompting the user.

If the client successfully authenticates by any of the methods listed above, it will attempt to cache the credentials on disk (unless the user has disabled this behavior, as mentioned earlier).

[2] Again, a common mistake is to misconfigure a server so that it never issues an authentication challenge. When users pass `--username` and `--password` options to the client, they're surprised to see that they're never used, i.e. new revisions still appear to have been committed anonymously!

```
$ svnserve -i
( success ( 1 2 ( ANONYMOUS ) ( edit-pipeline ) ) )
```

When invoked with the `--inetd` option, **svnserve** attempts to speak with a Subversion client via *stdin* and *stdout* using a custom protocol. This is the standard behavior for a program being run via **inetd**. The IANA has reserved port 3690 for the Subversion protocol, so on a Unix-like system you can add lines to `/etc/services` like these (if they don't already exist):

```
svn           3690/tcp   # Subversion
svn           3690/udp   # Subversion
```

And if your system is using a classic Unix-like **inetd** daemon, you can add this line to `/etc/inetd.conf`:

```
svn stream tcp nowait svnowner /usr/bin/svnserve svnserve -i
```

Make sure “svnowner” is a user which has appropriate permissions to access your repositories. Now, when a client connection comes into your server on port 3690, **inetd** will spawn an **svnserve** process to service it.

On a Windows system, third-party tools exist to run **svnserve** as a service. Look on Subversion's website for a list of these tools.

A second option is to run **svnserve** as a standalone “daemon” process. Use the `-d` option for this:

```
$ svnserve -d
$               # svnserve is now running, listening on port 3690
```

When running **svnserve** in daemon mode, you can use the `--listen-port=` and `--listen-host=` options to customize the exact port and hostname to “bind” to.

There's still a third way to invoke **svnserve**, and that's in “tunnel mode”, with the `-t` option. This mode assumes that a remote-service program such as **RSH** or **SSH** has successfully authenticated a user and is now invoking a private **svnserve** process *as that user*. The **svnserve** program behaves normally (communicating via *stdin* and *stdout*), and assumes that the traffic is being automatically redirected over some sort of tunnel back to the client. When **svnserve** is invoked by a tunnel agent like this, be sure that the authenticated user has full read and write access to the repository database files. (See Servers and Permissions: A Word of Warning.) It's essentially the same as a local user accessing the repository via `file:///` URLs.

Servers and Permissions: A Word of Warning

First, remember that a Subversion repository is a collection of database files; any process which accesses the repository directly needs to have proper read and write permissions on the entire repository. If you're not careful, this can lead to a number of headaches, especially if you're using a Berkeley DB database rather than FSFS. Be sure to read the section called “Supporting Multiple Repository Access Methods”.

Secondly, when configuring **svnserve**, Apache **httpd**, or any other server process, keep in mind that you might not want to launch the server process as the user `root` (or as any other user with unlimited permissions). Depending on the ownership and permissions of the repositories you're exporting, it's often prudent to use a different—perhaps custom—user. For example, many administrators create a new user named `svn`, grant that user exclusive ownership and rights to the exported Subversion repositories, and only run their server processes as that user.

Once the **svnserve** program is running, it makes every repository on your system available to the network. A client needs to specify an *absolute* path in the repository URL. For example, if a repository is located at `/usr/local/repositories/project1`, then a client would reach it via `svn://host.example.com/usr/local/repositories/project1` . To increase security, you can pass the `-r` option to **svnserve**, which restricts it to exporting only repositories below that path:

```
$ svnserve -d -r /usr/local/repositories
…
```

Using the `-r` option effectively modifies the location that the program treats as the root of the remote filesystem space. Clients then use URLs that have that path portion removed from them, leaving much shorter (and much less revealing) URLs:

```
$ svn checkout svn://host.example.com/project1
…
```

Built-in authentication and authorization

When a client connects to an **svnserve** process, the following things happen:

- The client selects a specific repository.
- The server processes the repository's `conf/svnserve.conf` file, and begins to enforce any authentication and authorization policies defined therein.
- Depending on the situation and authorization policies,
  - the client may be allowed to make requests anonymously, without ever receiving an authentication challenge, OR
  - the client may be challenged for authentication at any time, OR
  - if operating in “tunnel mode”, the client will declare itself to be already externally authenticated.

At the time of writing, the server only knows how to send a CRAM-MD5

[3]See RFC 2195.

ets ([and]), comments begin with hashes (#), and each section contains specific variables that can be set (variable = value). Let's walk through this file and learn how to use them.

Create a 'users' file and realm

For now, the [general] section of the svnserve.conf has all the variables you need. Begin by defining a file which contains usernames and passwords, and an authentication realm:

```
[general]
password-db = userfile
realm = example realm
```

The realm is a name that you define. It tells clients which sort of “authentication namespace” they're connecting to; the Subversion client displays it in the authentication prompt, and uses it as a key (along with the server's hostname and port) for caching credentials on disk (see the section called “Client Credentials Caching”). The password-db variable points to a separate file that contains a list of usernames and passwords, using the same familiar format. For example:

```
[users]
harry = foopassword
sally = barpassword
```

The value of password-db can be an absolute or relative path to the users file. For many admins, it's easy to keep the file right in the conf/ area of the repository, alongside svnserve.conf. On the other hand, it's possible you may want to have two or more repositories share the same users file; in that case, the file should probably live in a more public place. The repositories sharing the users file should also be configured to have the same realm, since the list of users essentially defines an authentication realm. Wherever the file lives, be sure to set the file's read and write permissions appropriately. If you know which user(s) **svnserve** will run as, restrict read access to the user file as necessary.

Set access controls

There are two more variables to set in the svnserve.conf file: they determine what unauthenticated (anonymous) and authenticated users are allowed to do. The variables anon-access and auth-access can be set to the values none, read, or write. Setting the value to none restricts all access of any kind; read allows read-only access to the repository, and write allows complete read/write access to the repository. For example:

```
[general]
password-db = userfile
realm = example realm

# anonymous users can only read the repository
anon-access = read

# authenticated users can both read and write
auth-access = write
```

The example settings are, in fact, the default values of the variables, should you forget to define them. If you want to be even more conservative, you can block anonymous access completely:

```
[general]
password-db = userfile
realm = example realm
```

```
# anonymous users aren't allowed
anon-access = none

# authenticated users can both read and write
auth-access = write
```

Notice that **svnserve** only understands “blanket” access control. A user either has universal read/write access, universal read access, or no access. There is no detailed control over access to specific paths within the repository. For many projects and sites, this level of access control is more than adequate. However, if you need per-directory access control, you'll need to use either use Apache with **mod_authz_svn** (see the section called “Per-Directory Access Control”) or use a **pre-commit** hook script to control write access (see the section called “Hook Scripts”). The Subversion distribution comes with **commit-access-control.pl** and the more sophisticated **svnperms.py** scripts for use in pre-commit scripts.

SSH authentication and authorization

svnserve's built-in authentication can be very handy, because it avoids the need to create real system accounts. On the other hand, some administrators already have well-established SSH authentication frameworks in place. In these situations, all of the project's users already have system accounts and the ability to “SSH into” the server machine.

It's easy to use SSH in conjunction with **svnserve**. The client simply uses the `svn+ssh://` URL schema to connect:

```
$ whoami
harry

$ svn list svn+ssh://host.example.com/repos/project
harry@host.example.com's password:  *****

foo
bar
baz
…
```

In this example, the Subversion client is invoking a local **ssh** process, connecting to `host.example.com`, authenticating as the user `harry`, then spawning a private **svnserve** process on the remote machine running as the user `harry`. The **svnserve** command is being invoked in tunnel mode (`-t`) and its network protocol is being “tunneled” over the encrypted connection by **ssh**, the tun-

When running over a tunnel, authorization is primarily controlled by operating system permissions to the repository's database files; it's very much the same as if Harry were accessing the repository directly via a `file:///` URL. If multiple system users are going to be accessing the repository directly, you may want to place them into a common group, and you'll need to be careful about umasks. (Be sure to read the section called “Supporting Multiple Repository Access Methods”.) But even in the case of tunneling, the `svnserve.conf` file can still be used to block access, by simply setting `auth-access = read` or `auth-access = none`.

You'd think that the story of SSH tunneling would end here, but it doesn't. Subversion allows you to create custom tunnel behaviors in your run-time `config` file (see the section called “Runtime Configuration Area”). For example, suppose you want to use RSH instead of SSH. In the `[tunnels]` section of your `config` file, simply define it like this:

```
[tunnels]
rsh = rsh
```

And now, you can use this new tunnel definition by using a URL schema that matches the name of your new variable: `svn+rsh://host/path`. When using the new URL schema, the Subversion client will actually be running the command **rsh host svnserve -t** behind the scenes. If you include a username in the URL (for example, `svn+rsh://username@host/path`) the client will also include that in its command (**rsh username@host svnserve -t**). But you can define new tunneling schemes to be much more clever than that:

```
[tunnels]
joessh = $JOESSH /opt/alternate/ssh -p 29934
```

This example demonstrates a couple of things. First, it shows how to make the Subversion client launch a very specific tunneling binary (the one located at `/opt/alternate/ssh`) with specific options. In this case, accessing a `svn+joessh://` URL would invoke the particular SSH binary with `-p 29934` as arguments—useful if you want the tunnel program to connect to a non-standard port.

Second, it shows how to define a custom environment variable that can override the name of the tunneling program. Setting the `SVN_SSH` environment variable is a convenient way to override the default SSH tunnel agent. But if you need to have several different overrides for different servers, each perhaps contacting a different port or passing a different set of options to SSH, you can use the mechanism demonstrated in this example. Now if we were to set the `JOESSH` environment variable, its value would override the entire value of the tunnel variable—**$JOESSH** would be executed instead of **/opt/alternate/ssh -p 29934**.

SSH configuration tricks

It's not only possible to control the way in which the client invokes **ssh**, but also to control the behavior of **sshd** on your server machine. In this section, we'll show how to control the exact **svnserve** command executed by **sshd**, as well as how to have multiple users share a single system account.

Initial setup

The lines are typically of the form:

```
ssh-dsa AAAABtce9euch.... user@example.com
```

The first field describes the type of key, the second field is the uuencoded key itself, and the third field is a comment. However, it's a lesser known fact that the entire line can be preceded by a `command` field:

```
command="program" ssh-dsa AAAABtce9euch.... user@example.com
```

When the `command` field is set, the SSH daemon will run the named program instead of the typical **svnserve -t** invocation that the Subversion client asks for. This opens the door to a number of server-side tricks. In the following examples, we abbreviate the lines of the file as:

```
command="program" TYPE KEY COMMENT
```

Controlling the invoked command

Because we can specify the executed server-side command, it's easy to name a specific **svnserve** binary to run and to pass it extra arguments:

```
command="/path/to/svnserve -t -r /virtual/root" TYPE KEY COMMENT
```

In this example, `/path/to/svnserve` might be a custom wrapper script around **svnserve** which sets the umask (see the section called “Supporting Multiple Repository Access Methods”). It also shows how to anchor **svnserve** in a virtual root directory, just as one often does when running **svnserve** as a daemon process. This might be done either to restrict access to parts of the system, or simply to relieve the user of having to type an absolute path in the `svn+ssh://` URL.

It's also possible to have multiple users share a single account. Instead of creating a separate system account for each user, generate a public/private keypair for each person. Then place each public key into the `authorized_users` file, one per line, and use the `--tunnel-user` option:

```
command="svnserve -t --tunnel-user=harry" TYPE1 KEY1 harry@example.com
command="svnserve -t --tunnel-user=sally" TYPE2 KEY2 sally@example.com
```

This example allows both Harry and Sally to connect to the same account via public-key authentication. Each of them has a custom command that will be executed; the `--tunnel-user` option tells **svnserve -t** to assume that the named argument is the authenticated user. Without `--tunnel-user`, it would appear as though all commits were coming from the one shared system account.

A final word of caution: giving a user access to the server via public-key in a shared account might still allow other forms of SSH access, even if you've set the `command` value in `authorized_keys`. For example, the user may still get shell access through SSH, or be able to perform X11 or general port-forwarding through your server. To give the user as little permission as possible, you may want to specify a number of restrictive options immediately after the `command`:

```
command="svnserve -t --tunnel-user=harry",no-port-forwarding,\
         no-agent-forwarding,no-X11-forwarding,no-pty \
         TYPE1 KEY1 harry@example.com
```

httpd, the Apache HTTP server

[4]They really hate doing that.

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

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

Basic Apache Configuration

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

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

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

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

At a later location in your configuration file, you now need to tell Apache where you keep your Subversion repository (or repositories). The `Location` directive has an XML-like notation, starting with an opening tag, and ending with a closing tag, with various other configuration directives in the middle. The purpose of the `Location` directive is to instruct Apache to do something special when handling requests that are directed at a given URL or one of its children. In the case of Subversion, you want Apache to simply hand off support for URLs that point at versioned resources to the DAV layer. You can instruct Apache to delegate the handling of all URLs whose path portions (the part of the URL that follows the server's name and the optional port number) begin with `/repos/` to a DAV provider whose repository is located at `/absolute/path/to/repository` using the following `httpd.conf` syntax:

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

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

```
<Location /svn>
  DAV svn

  # any "/svn/foo" URL will map to a repository /usr/local/svn/foo
  SVNParentPath /usr/local/svn
</Location>
```

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

Be sure that when you define your new `Location`, it doesn't overlap with other exported Locations. For example, if your main `DocumentRoot` is `/www`, do not export a Subversion repository in `<Location /www/repos>`. If a request comes in for the URI `/www/repos/foo.c`, Apache won't know whether to look for a file `repos/foo.c` in the `DocumentRoot`, or whether to delegate **mod_dav_svn** to return `foo.c` from the Subversion repository.

Server Names and the COPY Request

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

```
ServerName svn.example.com
```

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

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

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

Authentication Options

At this point, if you configured `httpd.conf` to contain something like

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

...then your repository is “anonymously” accessible to the world. Until you configure some authentication and authorization policies, the Subversion repositories you make available via the `Location` directive will be generally accessible to everyone. In other words,

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

Basic HTTP Authentication

The easiest way to authenticate a client is via the HTTP Basic authentication mechanism, which simply uses a username and password to verify that a user is who she says she is. Apache provides an **htpasswd** utility for managing the list of acceptable usernames and passwords, those to whom you wish to grant special access to your Subversion repository. Let's grant commit access to Sally and Harry. First, we need to add them to the password file.

```
$ ### First time: use -c to create the file
$ ### Use -m to use MD5 encryption of the password, which is more secure
$ htpasswd -cm /etc/svn-auth-file harry
New password: *****
Re-type new password: *****
Adding password for user harry
$ htpasswd -m /etc/svn-auth-file sally
New password: *******
Re-type new password: *******
Adding password for user sally
$
```

Next, you need to add some more `httpd.conf` directives inside your `Location` block to tell Apache what to do with your new password file. The `AuthType` directive specifies the type of authentication system to use. In this case, we want to specify the `Basic` authentication system. `AuthName` is an arbit-

This `<Location>` block is not yet complete, and will not do anything useful. It's merely telling Apache that whenever authorization is required, Apache should harvest a username and password from the Subversion client. What's missing here, however, are directives that tell Apache *which* sorts of client requests require authorization. Wherever authorization is required, Apache will demand authentication as well. The simplest thing to do is protect all requests. Adding `Require valid-user` tells Apache that all requests require an authenticated user:

```
<Location /svn>
  DAV svn
  SVNParentPath /usr/local/svn
  AuthType Basic
  AuthName "Subversion repository"
  AuthUserFile /etc/svn-auth-file
  Require valid-user
</Location>
```

Be sure to read the next section (the section called “Authorization Options”) for more detail on the `Require` directive and other ways to set authorization policies.

One word of warning: HTTP Basic Auth passwords pass in very nearly plain-text over the network, and thus are extremely insecure. If you're worried about password snooping, it may be best to use some sort of SSL encryption, so that clients authenticate via `https://` instead of `http://`; at a bare minimum, you can configure Apache to use a self-signed server certificate. [5] Consult Apache's documentation (and OpenSSL documentation) about how to do that.

SSL Certificate Management

Businesses that need to expose their repositories for access outside the company firewall should be conscious of the possibility that unauthorized parties could be “sniffing” their network traffic. SSL makes that kind of unwanted attention less likely to result in sensitive data leaks.

If a Subversion client is compiled to use OpenSSL, then it gains the ability to speak to an Apache server via `https://` URLs. The Neon library used by the Subversion client is not only able to verify server certificates, but can also supply client certificates when challenged. When the client and server have exchanged SSL certificates and successfully authenticated one another, all further communication is encrypted via a session key.

It's beyond the scope of this book to describe how to generate client and server certificates, and how to configure Apache to use them. Many other books, including Apache's own documentation, describe this task. But what *can* be covered here is how to manage server and client certificates from an ordinary Subversion client.

When speaking to Apache via `https://`, a Subversion client can receive two different types of information:

- a server certificate
- a demand for a client certificate

If the client receives a server certificate, it needs to verify that it trusts the certificate: is the server really who it claims to be? The OpenSSL library does this by examining the signer of the server certificate, or *certifying authority* (CA). If OpenSSL is unable to automatically trust the CA, or if some other problem occurs (such as an expired certificate or hostname mismatch), the Subversion command-line client will

[5] While self-signed server certificates are still vulnerable to a “man in the middle” attack, such an attack is still much more difficult for a casual observer to pull off, compared to sniffing unprotected passwords.

ask you whether you want to trust the server certificate anyway:

```
$ svn list https://host.example.com/repos/project

Error validating server certificate for 'https://host.example.com:443':
 - The certificate is not issued by a trusted authority. Use the
   fingerprint to validate the certificate manually!
Certificate information:
 - Hostname: host.example.com
 - Valid: from Jan 30 19:23:56 2004 GMT until Jan 30 19:23:56 2006 GMT
 - Issuer: CA, example.com, Sometown, California, US
 - Fingerprint: 7d:e1:a9:34:33:39:ba:6a:e9:a5:c4:22:98:7b:76:5c:92:a0:9c:7b

(R)eject, accept (t)emporarily or accept (p)ermanently?
```

This dialogue should look familiar; it's essentially the same question you've probably seen coming from your web browser (which is just another HTTP client like Subversion!). If you choose the (p)ermanent option, the server certificate will be cached in your private run-time `auth/`

```
[examplehost]
ssl-client-cert-file = /path/to/my/cert.p12
ssl-client-cert-password = somepassword
```

Once you've set the `ssl-client-cert-file` and `ssl-client-cert-password` variables, the Subversion client can automatically respond to a client certificate challenge without prompting you. [6]

Authorization Options

At this point, you've configured authentication, but not authorization. Apache is able to challenge clients and confirm identities, but it has not been told how to allow or restrict access to the clients bearing those identities. This section describes two strategies for controlling access to your repositories.

Blanket Access Control

The simplest form of access control is to authorize certain users for either read-only access to a repository, or read/write access to a repository.

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

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

  # how to authenticate a user
  AuthType Basic
  AuthName "Subversion repository"
  AuthUserFile /path/to/users/file

  # only authenticated users may access the repository
  Require valid-user
</Location>
```

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

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

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

[6] More security-conscious folk might not want to store the client certificate password in the runtime `servers` file.

```
  # how to authenticate a user
  AuthType Basic
  AuthName "Subversion repository"
  AuthUserFile /path/to/users/file

  # For any operations other than these, require an authenticated user.
  <LimitExcept GET PROPFIND OPTIONS REPORT>
    Require valid-user
  </LimitExcept>
</Location>
```

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

Per-Directory Access Control

It's possible to set up finer-grained permissions using a second Apache httpd module, **mod_authz_svn**. This module grabs the various opaque URLs passing from client to server, asks **mod_dav_svn** to decode them, and then possibly vetoes requests based on access policies defined in a configuration file.

If you've built Subversion from source code, **mod_authz_svn** is automatically built and installed alongside **mod_dav_svn**. Many binary distributions install it automatically as well. To verify that it's installed correctly, make sure it comes right after **mod_dav_svn**'s `LoadModule` directive in `httpd.conf`:

```
LoadModule dav_module         modules/mod_dav.so
LoadModule dav_svn_module     modules/mod_dav_svn.so
LoadModule authz_svn_module   modules/mod_authz_svn.so
```

To activate this module, you need to configure your `Location` block to use the `AuthzSVNAccessFile` directive, which specifies a file containing the permissions policy for paths within your repositories. (In a moment, we'll discuss the format of that file.)

Apache is flexible, so you have the option to configure your block in one of three general patterns. To begin, choose one of these basic configuration patterns. (The examples below are very simple; look at Apache's own documentation for much more detail on Apache authentication and authorization options.)

The simplest block is to allow open access to everyone. In this scenario, Apache never sends authentication challenges, so all users are treated as “anonymous”.

Example 6.1. A sample configuration for anonymous access.

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

  # our access control policy
  AuthzSVNAccessFile /path/to/access/file
</Location>
```

On the opposite end of the paranoia scale, you can configure your block to demand authentication from everyone. All clients must supply credentials to identify themselves. Your block unconditionally requires authentication via the `Require valid-user` directive, and defines a means to authenticate.

Example 6.2. A sample configuration for authenticated access.

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

  # our access control policy
  AuthzSVNAccessFile /path/to/access/file

  # only authenticated users may access the repository
  Require valid-user

  # how to authenticate a user
  AuthType Basic
  AuthName "Subversion repository"
  AuthUserFile /path/to/users/file
</Location>
```

A third very popular pattern is to allow a combination of authenticated and anonymous access. For example, many administrators want to allow anonymous users to read certain repository directories, but want only authenticated users to read (or write) more sensitive areas. In this setup, all users start out accessing the repository anonymously. If your access control policy demands a real username at any point, Apache will demand authentication from the client. To do this, you use both the `Satisfy Any` and `Require valid-user` directives together.

Example 6.3. A sample configuration for mixed authenticated/anonymous access.

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

  # our access control policy
  AuthzSVNAccessFile /path/to/access/file

  # try anonymous access first, resort to real
  # authentication if necessary.
  Satisfy Any
  Require valid-user

  # how to authenticate a user
  AuthType Basic
  AuthName "Subversion repository"
  AuthUserFile /path/to/users/file
</Location>
```

Once your basic `Location` block is configured, you can create an access file and define some authorization rules in it.

The syntax of the access file is the same familiar one used by **svnserve.conf** and the runtime configuration files. Lines that start with a hash (#) are ignored. In its simplest form, each section names a repository and path within it, and the authenticated usernames are the option names within each section. The value of each option describes the user's level of access to the repository path: either `r` (read-only) or

```
[/]
* = r
```

This is a common setup; notice that there's no repository name mentioned in the section name. This makes all repositories world readable to all users, whether you're using `SVNPath` or `SVNParentPath`. Once all users have read-access to the repositories, you can give explicit `rw` permission to certain users on specific subdirectories within specific repositories.

The asterisk variable (`*`) is also worth special mention here: it's the *only* pattern which matches an anonymous user. If you've configured your `Location` block to allow a mixture of anonymous and authenticated access, all users start out accessing Apache anonymously. **mod_authz_svn** looks for a `*` value defined for the path being accessed; if it can't find one, then Apache demands real authentication from the client.

(normally seen with the `--verbose` option), and the whole log message is suppressed. Needless to say, this can be time-consuming on revisions that affect a large number of files. This is the cost of security: even if you haven't configured a module like **mod_authz_svn** at all, the **mod_dav_svn** module is still asking Apache **httpd** to run authorization checks on every path. The **mod_dav_svn** module has no idea what authorization modules have been installed, so all it can do is ask Apache to invoke whatever might be present.

On the other hand, there's also an escape-hatch of sorts, one which allows you to trade security features for speed. If you're not enforcing any sort of per-directory authorization (i.e. not using **mod_authz_svn** or similar module), then you can disable all of this path-checking. In your `httpd.conf` file, use the `SVNPathAuthz` directive:

Example 6.4. Disabling path checks altogether

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

  SVNPathAuthz off
</Location>
```

The `SVNPathAuthz` directive is “on” by default. When set “off”, all path-based authorization checking is disabled; **mod_dav_svn** stops invoking authorization checks on every path it discovers.

Extra Goodies

We've covered most of the authentication and authorization options for Apache and mod_dav_svn. But there are a few other nice features that Apache provides.

Repository Browsing

One of the most useful benefits of an Apache/WebDAV configuration for your Subversion repository is that the youngest revisions of your versioned files and directories are immediately available for viewing via a regular web browser. Since Subversion uses URLs to identify versioned resources, those URLs used for HTTP-based repository access can be typed directly into a Web browser. Your browser will issue a `GET` request for that URL, and based on whether that URL represents a versioned directory or file, mod_dav_svn will respond with a directory listing or with file contents.

Since the URLs do not contain any information about which version of the resource you wish to see, mod_dav_svn will always answer with the youngest version. This functionality has the wonderful side-effect that you can pass around Subversion URLs to your peers as references to documents, and those URLs will always point at the latest manifestation of that document. Of course, you can even use the URLs as hyperlinks from other web sites, too.

You generally will get more use out of URLs to versioned files—after all, that's where the interesting content tends to lie. But you might have occasion to browse a Subversion directory listing, where you'll quickly note that the generated HTML used to display that listing is very basic, and certainly not intended to be aesthetically pleasing (or even interesting). To enable customization of these directory displays, Subversion provides an XML index feature. A single `SVNIndexXSLT` directive in your repository's `Location` block of `httpd.conf` will instruct mod_dav_svn to generate XML output when displaying a directory listing, and to reference the XSLT stylesheet of your choice:

```
<Location /svn>
  DAV svn
  SVNParentPath /usr/local/svn
  SVNIndexXSLT "/svnindex.xsl"
  …
</Location>
```

Using the SVNIndexXSLT directive and a creative XSLT stylesheet, you can make your directory listings match the color schemes and imagery used in other parts of your website. Or, if you'd prefer, you can use the sample stylesheets provided in the Subversion source distribution's tools/xslt/ directory. Keep in mind that the path provided to the SVNIndexXSLT directory is actually a URL path—browsers need to be able to read your stylesheets in order to make use of them!

Can I view older revisions?

With an ordinary web browser? In one word: nope. At least, not with **mod_dav_svn** as your only tool.

Your web browser only speaks ordinary HTTP. That means it only knows how to GET public URLs, which represent the latest versions of files and directories. According to the WebDAV/DeltaV spec, each server defines a private URL syntax for older versions of resources, and that syntax is opaque to clients. To find an older version of a file, a client must follow a specific procedure to “discover” the proper URL; the procedure involves issuing a series of WebDAV PROPFIND requests and understanding DeltaV concepts. This is something your web browser simply can't do.

So to answer the question, one obvious way to see older revisions of files and directories is by passing the --revision argument to the **svn list** and **svn cat** commands. To browse old revisions with your web browser, however, you can use third-party software. A good example of this is ViewVC (http://viewvc.tigris.org/). ViewVC was originally written to display CVS repositories through the web, [7] and the latest bleeding-edge versions (at the time of writing) are able to understand Subversion repositories as well.

Other Features

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

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

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

Finally, because **mod_dav_svn** is speaking a semi-complete dialect of WebDAV/DeltaV, it's possible to access the repository via third-party DAV clients. Most modern operating systems (Win32, OS X, and Linux) have the built-in ability to mount a DAV server as a standard network “share”. This is a complic-

[7] Back then, it was called “ViewCVS”.

ated topic; for details, read Appendix B, *WebDAV and Autoversioning*.

Supporting Multiple Repository Access Methods

You've seen how a repository can be accessed in many different ways. But is it possible—or safe—for your repository to be accessed by multiple methods simultaneously? The answer is yes, provided you use a bit of foresight.

At any given time, these processes may require read and write access to your repository:

- regular system users using a Subversion client (as themselves) to access the repository directly via `file:///` URLs;
- regular system users connecting to SSH-spawned private **svnserve** processes (running as themselves) which access the repository;
- an **svnserve** process—either a daemon or one launched by **inetd**—running as a particular fixed user;
- an Apache **httpd** process, running as a particular fixed user.

The most common problem administrators run into is repository ownership and permissions. Does every process (or user) in the previous list have the rights to read and write the Berkeley DB files? Assuming you have a Unix-like operating system, a straightforward approach might be to place every potential repository user into a new `svn` group, and make the repository wholly owned by that group. But even that's not enough, because a process may write to the database files using an unfriendly umask—one that prevents access by other users.

So the next step beyond setting up a common group for repository users is to force every repository-accessing process to use a sane umask. For users accessing the repository directly, you can make the **svn** program into a wrapper script that first sets **umask 002** and then runs the real **svn** client program. You can write a similar wrapper script for the **svnserve** program, and add a **umask 002** command to Apache's own startup script, `apachectl`. For example:

```
$ cat /usr/bin/svn

#!/bin/sh

umask 002
/usr/bin/svn-real "$@"
```

Another common problem is often encountered on Unix-like systems. As a repository is used, Berkeley DB occasionally creates new log files to journal its actions. Even if the repository is wholly owned by the **svn** group, these newly created files won't necessarily be owned by that same group, which then creates more permissions problems for your users. A good workaround is to set the group SUID bit on the repository's `db` directory. This causes all newly-created log files to have the same group owner as the parent directory.

Once you've jumped through these hoops, your repository should be accessible by all the necessary processes. It may seem a bit messy and complicated, but the problems of having multiple users sharing write-access to common files are classic ones that are not often elegantly solved.

Fortunately, most repository administrators will never *need* to have such a complex configuration. Users who wish to access repositories that live on the same machine are not limited to using `file://` access URLs—they can typically contact the Apache HTTP server or **svnserve** using `localhost` for the server name in their `http://` or `svn://` URLs. And to maintain multiple server processes for your Subversion repositories is likely to be more of a headache than necessary. We recommend you choose the server that best meets your needs and stick with it!

The svn+ssh:// server checklist

It can be quite tricky to get a bunch of users with existing SSH accounts to share a repository without permissions problems. If you're confused about all the things that you (as an administrator) need to do on a Unix-like system, here's a quick checklist that resummarizes some of things discussed in this section:

- All of your SSH users need to be able to read and write to the repository. Put all the SSH users into a single group. Make the repository wholly owned by that group, and set the group permissions to read/write.
- Your users need to use a sane umask when accessing the repository. Make sure that **svnserve** (`/usr/bin/svnserve`, or wherever it lives in `$PATH`) is actually a wrapper script which sets **umask 002** and executes the real **svnserve** binary. Take similar measures when using **svnlook** and **svnadmin**. Either run them with a sane umask, or wrap them as described above.

location (again, as specified by the Windows Registry). Unlike the per-user case, the **svn** program does not attempt to create the system-wide configuration area.

The configuration area currently contains three files—two configuration files (`config` and `servers`), and a `README.txt` file which describes the INI format. At the time of their creation, the files contain default values for each of the supported Subversion options, mostly commented out and grouped with textual descriptions about how the values for the key affect Subversion's behavior. To change a certain behavior, you need only to load the appropriate configuration file into a text editor, and modify the desired option's value. If at any time you wish to have the default configuration settings restored, you can simply remove (or rename) your configuration directory and then run some innocuous **svn** command, such as **svn --version**. A new configuration directory with the default contents will be created.

The per-user configuration area also contains a cache of authentication data. The `auth` directory holds a set of subdirectories that contain pieces of cached information used by Subversion's various supported authentication methods. This directory is created in such a way that only the user herself has permission to read its contents.

Configuration and the Windows Registry

In addition to the usual INI-based configuration area, Subversion clients running on Windows platforms may also use the Windows registry to hold the configuration data. The option names and their values are the same as in the INI files. The "file/section" hierarchy is preserved as well, though addressed in a slightly different fashion—in this schema, files and sections are just levels in the registry key tree.

Subversion looks for system-wide configuration values under the `HKEY_LOCAL_MACHINE\Software\Tigris.org\Subversion` key. For example, the `global-ignores` option, which is in the `miscellany` section of the `config` file, would be found at `HKEY_LOCAL_MACHINE\Software\Tigris.org\Subversion\Config\Miscellany\global-ignores`. Per-user configuration values should be stored under `HKEY_CURRENT_USER\Software\Tigris.org\Subversion`.

Registry-based configuration options are parsed *before* their file-based counterparts, so are overridden by values found in the configuration files. In other words, configuration priority is granted in the following order on a Windows system:

1. Command-line options

2. The per-user INI files

```
REGEDIT4

[HKEY_LOCAL_MACHINE\Software\Tigris.org\Subversion\Servers\groups]

[HKEY_LOCAL_MACHINE\Software\Tigris.org\Subversion\Servers\global]
"#http-proxy-host"=""
"#http-proxy-port"=""
"#http-proxy-username"=""
"#http-proxy-password"=""
"#http-proxy-exceptions"=""
"#http-timeout"="0"
"#http-compression"="yes"
"#neon-debug-mask"=""
"#ssl-authority-files"=""
"#ssl-trust-default-ca"=""
"#ssl-client-cert-file"=""
"#ssl-client-cert-password"=""

[HKEY_CURRENT_USER\Software\Tigris.org\Subversion\Config\auth]
"#store-auth-creds"="no"

[HKEY_CURRENT_USER\Software\Tigris.org\Subversion\Config\helpers]
"#editor-cmd"="notepad"
"#diff-cmd"=""
"#diff3-cmd"=""
"#diff3-has-program-arg"=""

[HKEY_CURRENT_USER\Software\Tigris.org\Subversion\Config\miscellany]
"#global-ignores"="*.o *.lo *.la #*# .*.rej *.rej .*~ *~ .#* .DS_Store"
"#log-encoding"=""
"#use-commit-times"=""
"#template-root"=""
"#enable-auto-props"=""

[HKEY_CURRENT_USER\Software\Tigris.org\Subversion\Config\tunnels]

[HKEY_CURRENT_USER\Software\Tigris.org\Subversion\Config\auto-props]
```

The previous example shows the contents of a `.reg` file which contains some of the most commonly used configuration options and their default values. Note the presence of both system-wide (for network proxy-related options) and per-user settings (editor programs and password storage, among others). Also note that all the options are effectively commented out. You need only to remove the hash (#) character from the beginning of the option names, and set the values as you desire.

Configuration Options

In this section, we will discuss the specific run-time configuration options that are currently supported by Subversion.

Servers

The `servers` file contains Subversion configuration options related to the network layers. There are two special section names in this file—`groups` and `global`. The `groups` section is essentially a cross-reference table. The keys in this section are the names of other sections in the file; their values are *globs*—textual tokens which possibly contain wildcard characters—that are compared against the hostnames of the machine to which Subversion requests are sent.

```
[groups]
beanie-babies = *.red-bean.com
collabnet = svn.collab.net

[beanie-babies]
…

[collabnet]
…
```

When Subversion is used over a network, it attempts to match the name of the server it is trying to reach with a group name under the `groups` section. If a match is made, Subversion then looks for a section in the `servers` file whose name is the matched group's name. From that section it reads the actual network configuration settings.

The `global` section contains the settings that are meant for all of the servers not matched by one of the globs under the `groups` section. The options available in this section are exactly the same as those valid for the other server sections in the file (except, of course, the special `groups` section), and are as follows:

`http-proxy-host`
This specifies the hostname of the proxy computer through which your HTTP-based Subversion requests must pass. It defaults to an empty value, which means that Subversion will not attempt to route HTTP requests through a proxy computer, and will instead attempt to contact the destination machine directly.

`http-proxy-port`
This specifies the port number on the proxy host to use. It defaults to an empty value.

`http-proxy-username`
This specifies the username to supply to the proxy machine. It defaults to an empty value.

`http-proxy-password`
This specifies the password to supply to the proxy machine. It defaults to an empty value.

`http-timeout`
This specifies the amount of time, in seconds, to wait for a server response. If you experience problems with a slow network connection causing Subversion operations to timeout, you should increase the value of this option. The default value is `0`, which instructs the underlying HTTP library, Neon, to use its default timeout setting.

`http-compression`
This specifies whether or not Subversion should attempt to compress network requests made to DAV-ready servers. The default value is `yes` (though compression will only occur if that capability is compiled into the network layer). Set this to `no` to disable compression, such as when debugging network transmissions.

`neon-debug-mask`
This is an integer mask that the underlying HTTP library, Neon, uses for choosing what type of debugging output to yield. The default value is `0`, which will silence all debugging output. For more information about how Subversion makes use of Neon, see Chapter 8, *Developer Information*.

`ssl-authority-files`
This is a semicolon-delimited list of paths to files containing certificates of the certificate authorities (or CAs) that are accepted by the Subversion client when accessing the repository over HTTPS.

`ssl-trust-default-ca`
Set this variable to `yes` if you want Subversion to automatically trust the set of default CAs that ship with OpenSSL.

ssl-client-cert-file

should modify the temporary file and return a zero exit code to indicate success.

`diff-cmd`
This specifies the absolute path of a differencing program, used when Subversion generates "diff" output (such as when using the **svn diff** command). By default Subversion uses an internal differencing library—setting this option will cause it to perform this task using an external program. See the section called "Using External Differencing Tools" for more details on using such programs.

`diff3-cmd`
This specifies the absolute path of a three-way differencing program. Subversion uses this program to merge changes made by the user with those received from the repository. By default Subversion uses an internal differencing library—setting this option will cause it to perform this task using an external program. See the section called "Using External Differencing Tools" for more details on using such programs.

`diff3-has-program-arg`
This flag should be set to `true` if the program specified by the `diff3-cmd` option accepts a `--diff-program` command-line parameter.

The `tunnels` section allows you to define new tunnel schemes for use with **svnserve** and `svn://` client connections. For more details, see the section called "SSH authentication and authorization".

The `miscellany` section is where everything that doesn't belong elsewhere winds up. [2] In this section, you can find:

`global-ignores`
When running the **svn status** command, Subversion lists unversioned files and directories along with the versioned ones, annotating them with a `?` character (see the section called "**svn status**"). Sometimes, it can be annoying to see uninteresting, unversioned items—for example, object files that result from a program's compilation—in this display. The `global-ignores` option is a list of whitespace-delimited globs which describe the names of files and directories that Subversion should not display unless they are versioned. The default value is `*.o *.lo *.la #*# .*.rej *.rej .*~ *~ .#* .DS_Store`.

As well as **svn status**, the **svn add** and **svn import** commands also ignore files that match the list when they are scanning a directory. You can override this behaviour for a single instance of any of these commands by explicitly specifying the file name, or by using the `--no-ignore` command-line flag.

For information on more fine-grained control of ignored items, see the section called "`svn:ignore`".

`enable-auto-props`
This instructs Subversion to automatically set properties on newly added or imported files. The default value is `no`, so set this to `yes` to enable Auto-props. The `auto-props` section of this file specifies which properties are to be set on which files.

`log-encoding`
This variable sets the default character set encoding for commit log messages. It's a permanent form of the `--encoding` option (see the section called "**svn** Switches"). The Subversion repository stores log messages in UTF-8, and assumes that your log message is written using your operating system's native locale. You should specify a different encoding if your commit messages are written in any other encoding.

`use-commit-times`
Normally your working copy files have timestamps that reflect the last time they were touched by any process, whether that be your own editor or by some **svn** subcommand. This is generally convenient for people developing software, because build systems often look at timestamps as a way of deciding which files need to be recompiled.

In other situations, however, it's sometimes nice for the working copy files to have timestamps that reflect the

[2] Anyone for potluck dinner?

last time they were changed in the repository. The **svn export** command always places these “last-commit timestamps” on trees that it produces. By setting this config variable to `yes`, the **svn checkout**, **svn update**, **svn switch**, and **svn revert** commands will also set last-commit timestamps on files that they touch.

The `auto-props` section controls the Subversion client's ability to automatically set properties on files when they are added or imported. It contains any number of key-value pairs in the format `PATTERN = PROPNAME=PROPVALUE` where `PATTERN` is a file pattern that matches a set of filenames and the rest of the line is the property and its value. Multiple matches on a file will result in multiple propsets for that file; however, there is no guarantee that auto-props will be applied in the order in which they are listed in the config file, so you can't have one rule “override” another. You can find several examples of auto-props usage in the `config` file. Lastly, don't forget to set `enable-auto-props` to `yes` in the `miscellany` section if you want to enable auto-props.

Properties

We've already covered in detail how Subversion stores and retrieves various versions of files and directories in its repository. Whole chapters have been devoted to this most fundamental piece of functionality provided by the tool. And if the versioning support stopped there, Subversion would still be complete from a version control perspective. But it doesn't stop there.

In addition to versioning your directories and files, Subversion provides interfaces for adding, modifying, and removing versioned metadata on each of your versioned directories and files. We refer to this metadata as *properties*, and they can be thought of as two-column tables that map property names to arbitrary values attached to each item in your working copy. Generally speaking, the names and values of the properties can be whatever you want them to be, with the constraint that the names must be human-readable text. And the best part about these properties is that they, too, are versioned, just like the textual contents of your files. You can modify, commit, and revert property changes as easily as committing textual changes. And you receive other people's property changes as you update your working copy.

Other Properties in Subversion

Properties show up elsewhere in Subversion, too. Just as files and directories may have arbitrary property names and values attached to them, each revision as a whole may have arbitrary properties attached to it. The same constraints apply—human-readable, text names and anything-you-want, binary values—except that revision properties are not versioned. See the section called “Unversioned Properties” for more information on these unversioned properties.

In this section, we will examine the utility—both to users of Subversion, and to Subversion itself—of property support. You'll learn about the property-related **svn** subcommands, and how property modifications affect your normal Subversion workflow. Hopefully, you'll be convinced that Subversion properties can enhance your version control experience.

Why Properties?

Properties can be very useful additions to your working copy. In fact, Subversion itself uses properties to house special information, and as a way to denote that certain special processing might be needed. Likewise, you can use properties for your own purposes. Of course, anything you can do with properties you could also do using regular versioned files, but consider the following example of Subversion property use.

Say you wish to design a website that houses many digital photos, and displays them with captions and a datestamp. Now, your set of photos is constantly changing, so you'd like to have as much of this site automated as possible. These photos can be quite large, so as is common with sites of this nature, you want to provide smaller thumbnail images to your site visitors. You can do this with traditional files. That is, you can have your `image123.jpg` and an `image123-thumbnail.jpg` side-by-side in a directory. Or if you want to keep the filenames the same, you might have your thumbnails in a different directory, like `thumbnails/image123.jpg`. You can also store your captions and datestamps in a similar fashion, again separated from the original image file. Soon, your tree of files is a mess, and grows in multiples with each new photo added to the site.

Now consider the same setup using Subversion's file properties. Imagine having a single image file, `image123.jpg`, and then properties set on that file named `caption`, `datestamp`, and even `thumbnail`. Now your working copy directory looks much more manageable—in fact, it looks like there are nothing but image files in it. But your automation scripts know better. They know that they can use **svn** (or better yet, they can use the Subversion language bindings—see the section called “Using Languages Other than C and C++”) to dig out the extra information that your site needs to display without having to read an index file or play path manipulation games.

How (and if) you use Subversion properties is up to you. As we mentioned, Subversion has it own uses for properties, which we'll discuss a little later in this chapter. But first, let's discuss how to manipulate properties using the **svn** program.

Manipulating Properties

The **svn** command affords a few ways to add or modify file and directory properties. For properties with short, human-readable values, perhaps the simplest way to add a new property is to specify the property name and value on the command-line of the **propset** subcommand.

```
$ svn propset copyright '(c) 2003 Red-Bean Software' calc/button.c
property 'copyright' set on 'calc/button.c'
$
```

But we've been touting the flexibility that Subversion offers for your property values. And if you are planning to have a multi-line textual, or even binary, property value, you probably do not want to supply that value on the command-line. So the **propset** subcommand takes a `--file` (`-F`) option for specifying the name of a file which contains the new property value.

```
$ svn propset license -F /path/to/LICENSE calc/button.c
property 'license' set on 'calc/button.c'
$
```

There are some restrictions on the names you can use for properties. A property name must start with a letter, a colon (`:`), or an underscore (`_`); after that, you can also use digits, hyphens (`-`), and periods (`.`). [3]

In addition to the **propset** command, the **svn** program supplies the **propedit** command. This command uses the configured editor program (see the section called “Config”) to add or modify properties. When you run the command, **svn** invokes your editor program on a temporary file that contains the current value of the property (or which is empty, if you are adding a new property). Then, you just modify that value in your editor program until it represents the new value you wish to store for the property, save the temporary file, and then exit the editor program. If Subversion detects that you've actually changed the existing value of the property, it will accept that as the new property value. If you exit your editor without making any changes, no property modification will occur.

```
$ svn propedit copyright calc/button.c  ### exit the editor without changes
```

[3] If you're familiar with XML, this is pretty much the ASCII subset of the syntax for XML "Name".

```
No changes to property 'copyright' on 'calc/button.c'
$
```

We should note that, as with other **svn** subcommands, those related to properties can act on multiple paths at once. This enables you to modify properties on whole sets of files with a single command. For example, we could have done:

```
$ svn propset copyright '(c) 2002 Red-Bean Software' calc/*
property 'copyright' set on 'calc/Makefile'
property 'copyright' set on 'calc/button.c'
property 'copyright' set on 'calc/integer.c'
…
$
```

All of this property adding and editing isn't really very useful if you can't easily get the stored property value. So the **svn** program supplies two subcommands for displaying the names and values of properties stored on files and directories. The **svn proplist** command will list the names of properties that exist on a path. Once you know the names of the properties on the node, you can request their values individually using **svn propget**. This command will, given a path (or set of paths) and a property name, print the value of the property to the standard output stream.

```
$ svn proplist calc/button.c
Properties on 'calc/button.c':
  copyright
  license
$ svn propget copyright calc/button.c
(c) 2003 Red-Bean Software
```

There's even a variation of the **proplist** command that will list both the name and value of all of the properties. Simply supply the `--verbose` (`-v`) option.

```
$ svn proplist --verbose calc/button.c
Properties on 'calc/button.c':
  copyright : (c) 2003 Red-Bean Software
  license : ================================================================
Copyright (c) 2003 Red-Bean Software.  All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

1. Redistributions of source code must retain the above copyright
notice, this list of conditions, and the recipe for Fitz's famous
red-beans-and-rice.
…
```

The last property-related subcommand is **propdel**. Since Subversion allows you to store properties with empty values, you can't remove a property altogether using **propedit** or **propset**. For example, this command will *not* yield the desired effect:

```
$ svn propset license '' calc/button.c
property 'license' set on 'calc/button.c'
$ svn proplist --verbose calc/button.c
Properties on 'calc/button.c':
```

```
  copyright : (c) 2003 Red-Bean Software
  license :
$
```

You need to use the **propdel** command to delete properties altogether. The syntax is similar to the other property commands:

```
$ svn propdel license calc/button.c
property 'license' deleted from 'calc/button.c'.
$ svn proplist --verbose calc/button.c
Properties on 'calc/button.c':
  copyright : (c) 2003 Red-Bean Software
$
```

Now that you are familiar with all of the property-related **svn** subcommands, let's see how property modifications affect the usual Subversion workflow. As we mentioned earlier, file and directory properties are versioned, just like your file contents. As a result, Subversion provides the same opportunities for merging—in cleanly or conflicting fashions—someone else's modifications into your own.

Modifying Revision Properties

Remember those unversioned revision properties? You can modify those, too, with the **svn** program. Simply add the `--revprop` command-line parameter, and specify the revision whose property you wish to modify. Since revisions are global, you don't need to specify a path in this case as long as you are positioned in the working copy of the repository whose revision property you wish to modify. For example, you might want to replace the commit log message of an existing revision.[4]

```
$ svn propset svn:log '* button.c: Fix a compiler warning.' -r11 --revprop
property 'svn:log' set on repository revision '11'
$
```

Note that the ability to modify these unversioned properties must be explicitly added by the repository administrator (see the section called “Hook Scripts”). Since the properties aren't versioned, you run the risk of losing information if you aren't careful with your edits. The repository administrator can setup methods to protect against this loss, and by default, modification of unversioned properties is disabled.

And as with file contents, your property changes are local modifications, only made permanent when you commit them to the repository with **svn commit**. Your property changes can be easily unmade, too—the **svn revert** command will restore your files and directories to their un-edited states, contents, properties, and all. Also, you can receive interesting information about the state of your file and directory properties by using the **svn status** and **svn diff** commands.

```
$ svn status calc/button.c
 M     calc/button.c
$ svn diff calc/button.c
Property changes on: calc/button.c
___________________________________________________________________
Name: copyright
   + (c) 2003 Red-Bean Software
```

[4]Fixing spelling errors, grammatical gotchas, and “just-plain-wrongness” in commit log messages is perhaps the most common use case for the `--revprop` option.

```
$
```

Notice how the **status** subcommand displays `M` in the second column instead of the first. That is because we have modified the properties on `calc/button.c`, but not modified its textual contents. Had we changed both, we would have seen `M` in the first column, too (see the section called "**svn status**").

Property Conflicts

As with file contents, local property modifications can conflict with changes committed by someone else. If you update your working copy directory and receive property changes on a versioned resource that clash with your own, Subversion will report that the resource is in a conflicted state.

```
% svn update calc
M  calc/Makefile.in
 C calc/button.c
Updated to revision 143.
$
```

Subversion will also create, in the same directory as the conflicted resource, a file with a `.prej` extension which contains the details of the conflict. You should examine the contents of this file so you can decide how to resolve the conflict. Until the conflict is resolved, you will see a `C` in the second column of **svn status** output for that resource, and attempts to commit your local modifications will fail.

```
$ svn status calc
 C     calc/button.c
?      calc/button.c.prej
$ cat calc/button.c.prej
prop 'linecount': user set to '1256', but update set to '1301'.
$
```

To resolve property conflicts, simply ensure that the conflicting properties contain the values that they should, and then use the **svn resolved** command to alert Subversion that you have manually resolved the problem.

You might also have noticed 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. This does unfortunately mean that to fully apply a patch generated by **svn diff**, any property modifications will need to be applied by hand.

As you can see, the presence of property modifications has no outstanding effect on the typical Subversion workflow. Your general patterns of updating your working copy, checking the status of your files and directories, reporting on the modifications you have made, and committing those modifications to the repository are completely immune to the presence or absence of properties. The **svn** program has some additional subcommands for actually making property changes, but that is the only noticeable asymmetry.

Special Properties

Subversion has no particular policy regarding properties—you can use them for any purpose. Subversion asks only that you not use property names that begin with the prefix `svn:`. That's the namespace that it sets aside for its own use. In fact, Subversion defines certain properties that have magical effects on the files and directories to which they are attached. In this section, we'll untangle the mystery, and describe how these special properties make your life just a little easier.

svn:executable

The `svn:executable` property is used to control a versioned file's filesystem-level execute permission bit in a semi-automated way. This property has no defined values—its mere presence indicates a

[5]The Windows filesystems use file extensions (such as `.EXE`, `.BAT`, and `.COM`) to denote executable files.

working copy that is not already filtered out by the `global-ignores` option (or its built-in default value). This is done so that users can see if perhaps they've forgotten to add a resource to version control.

But Subversion cannot possibly guess the names of every resource that should be ignored. Also, quite often there are things that should be ignored in *every* working copy of a particular repository. To force every user of that repository to add patterns for those resources to their run-time configuration areas would be not just a burden, but has the potential to clash with the configuration needs of other working copies that the user has checked out.

The solution is to store ignore patterns that are unique to the resources likely to appear in a given directory with the directory itself. Common examples of unversioned resources that are basically unique to a directory, yet likely to appear there, include output from program compilations. Or—to use an example more appropriate to this book—the HTML, PDF, or PostScript files generated as the result of a conversion of some source DocBook XML files to a more legible output format.

Ignore Patterns for CVS Users

The Subversion `svn:ignore` property is very similar in syntax and function to the CVS `.cvsignore` file. In fact, if you are migrating a CVS working copy to Subversion, you can directly migrate the ignore patterns by using the `.cvsignore` file as input file to the **svn propset** command:

```
$ svn propset svn:ignore -F .cvsignore .
property 'svn:ignore' set on '.'
$
```

There are, however, some differences in the ways that CVS and Subversion handle ignore patterns. The two systems use the ignore patterns at some different times, and there are slight discrepancies in what the ignore patterns apply to. Also, Subversion does not recognize the use of the `!` pattern as a reset back to having no ignore patterns at all.

For this purpose, the `svn:ignore` property is the solution. Its value is a multi-line collection of file patterns, one pattern per line. The property is set on the directory in which you wish the patterns to be applied. [6] For example, say you have the following output from **svn status**:

```
$ svn status calc
 M     calc/button.c
?      calc/calculator
?      calc/data.c
?      calc/debug_log
?      calc/debug_log.1
?      calc/debug_log.2.gz
?      calc/debug_log.3.gz
```

In this example, you have made some property modifications to `button.c`, but in your working copy you also have some unversioned files: the latest `calculator` program that you've compiled from your source code, a source file named `data.c`, and a set of debugging output log files. Now, you know that your build system always results in the `calculator` program being generated. [7] And you know that your test suite always leaves those debugging log files lying around. These facts are true for all working copies, not just your own. And you know that you aren't interested in seeing those things every time you run **svn status**. So you use **svn propedit svn:ignore calc** to add some ignore patterns to the `calc` directory. For example, you might add this as the new value of the `svn:ignore` property:

[6]The patterns are strictly for that directory—they do not carry recursively into subdirectories.
[7]Isn't that the whole point of a build system?

```
calculator
debug_log*
```

After you've added this property, you will now have a local property modification on the `calc` directory. But notice what else is different about your **svn status** output:

```
$ svn status
 M     calc
 M     calc/button.c
?      calc/data.c
```

Now, all the cruft is missing from the output! Of course, those files are still in your working copy. Subversion is simply not reminding you that they are present and unversioned. And now with all the trivial noise removed from the display, you are left with more interesting items—such as that source code file that you probably forgot to add to version control.

If you want to see the ignored files, you can pass the `--no-ignore` option to Subversion:

```
$ svn status --no-ignore
 M     calc/button.c
I      calc/calculator
?      calc/data.c
I      calc/debug_log
I      calc/debug_log.1
I      calc/debug_log.2.gz
I      calc/debug_log.3.gz
```

The list of patterns to ignore is also used by **svn add** and **svn import**. Both of these operations involve asking Subversion to begin managing some set of files and directories. Rather than force the user to pick and choose which files in a tree she wishes to start versioning, Subversion uses the ignore patterns to determine which files should not be swept into the version control system as part of a larger recursive addition or import operation.

`svn:keywords`

Subversion has the ability to substitute *keywords*—pieces of useful, dynamic information about a versioned file—into the contents of the file itself. Keywords generally describe information about the last

Subversion defines the list of keywords available for substitution. That list contains the following five keywords, some of which have aliases that you can also use:

`Date`
This keyword describes the last time the file was known to have been changed in the repository, and looks something like `$Date: 2002-07-22 21:42:37 -0700 (Mon, 22 Jul 2002) $`. It may also be specified as `LastChangedDate`.

`Revision`
This keyword describes the last known revision in which this file changed in the repository, and looks something like `$Revision: 144 $`. It may also be specified as `LastChangedRevision` or `Rev`.

`Author`
This keyword describes the last known user to change this file in the repository, and looks something like `$Author: harry $`. It may also be specified as `LastChangedBy`.

`HeadURL`
This keyword describes the full URL to the latest version of the file in the repository, and looks something like `$HeadURL: http://svn.collab.net/repos/trunk/README $`. It may be abbreviated as `URL`.

`Id`
This keyword is a compressed combination of the other keywords. Its substitution looks something like `$Id: calc.c 148 2002-07-28 21:30:43Z sally $`, and is interpreted to mean that the file `calc.c` was last changed in revision 148 on the evening of July 28, 2002 by the user `sally`.

Simply adding keyword anchor text to your file does nothing special. Subversion will never attempt to perform textual substitutions on your file contents unless explicitly asked to do so. After all, you might be writing a document [8] about how to use keywords, and you don't want Subversion to substitute your beautiful examples of un-substituted keyword anchors!

To tell Subversion whether or not to substitute keywords on a particular file, we again turn to the property-related subcommands. The `svn:keywords` property, when set on a versioned file, controls which

[8]… or maybe even a section of a book …

erty value we set. Subversion will happily ignore requests to substitute keywords that are not present in the file, and will not substitute keywords that are not present in the `svn:keywords` property value.

Keywords and Spurious Differences

The user-visible result of keyword substitution might lead you to think that every version of a file with that feature in use differs from the previous version in at least the area where the keyword anchor was placed. However, this is actually not the case. While checking for local modifications during **svn diff**, and before transmitting those local modifications during **svn commit**, Subversion “un-substitutes” any keywords that it previously substituted. The result is that the versions of the file that are stored in the repository contain only the real modifications that users make to the file.

Immediately after you commit this property change, Subversion will update your working file with the new substitute text. Instead of seeing your keyword anchor `$LastChangedDate$`, you'll see its substituted result. That result also contains the name of the keyword, and continues to be bounded by the dollar sign (`$`) characters. And as we predicted, the `Rev` keyword was not substituted because we didn't ask for it to be.

Note also that we set the `svn:keywords` property to “Date Author” yet the keyword anchor used the alias `$LastChangedDate$` and still expanded correctly.

```
Here is the latest report from the front lines.
$LastChangedDate: 2002-07-22 21:42:37 -0700 (Mon, 22 Jul 2002) $
$Rev$
Cumulus clouds are appearing more frequently as summer approaches.
```

If someone else now commits a change to `weather.txt`, your copy of that file will continue to display the same substituted keyword value as before—until you update your working copy. At that time the keywords in your `weather.txt` file will be re-substituted with information that reflects the most recent known commit to that file.

Subversion 1.2 introduced a new variant of the keyword syntax which brought additional, useful—though perhaps atypical—functionality. You can now tell Subversion to maintain a fixed length (in terms of the number of bytes consumed) for the substituted keyword. By using a double-colon (`::`) after the keyword name, followed by a number of space characters, you define that fixed width. When Subversion goes to substitute your keyword for the keyword and its value, it will essentially replace only those space characters, leaving the overall width of the keyword field unchanged. If the substituted value is shorter than the defined field width, there will be extra padding characters (spaces) at the end of the substituted field; if it is too long, it is truncated with a special hash (#) character just before the final dollar sign terminator.

For example, say you have a document in which you have some section of tabular data reflecting the document's Subversion keywords. Using the original Subversion keyword substitution syntax, your file might look something like:

```
$Rev$:     Revision of last commit
$Author$:  Author of last commit
$Date$:    Date of last commit
```

Now, that looks nice and tabular at the start of things. But when you then commit that file (with keyword substitution enabled, of course), you see:

```
$Rev: 12 $:     Revision of last commit
```

```
$Author: harry $:  Author of last commit
$Date: 2006-03-15 02:33:03 -0500 (Wed, 15 Mar 2006) $:    Date of last commit
```

The result is not so beautiful. And you might be tempted to then adjust the file after the substitution so that it again looks tabular. But that only holds as long as the keyword values are the same width. If the last committed revision rolls into a new place value (say, from 99 to 100), or if another person with a longer username commits the file, stuff gets all crooked again. However, if you are using Subversion 1.2 or better, you can use the new fixed-length keyword syntax, define some field widths that seem sane, and now your file might look like this:

```
$Rev::               $:  Revision of last commit
$Author::            $:  Author of last commit
$Date::              $:  Date of last commit
```

You commit this change to your file. This time, Subversion notices the new fixed-length keyword syntax, and maintains the width of the fields as defined by the padding you placed between the double-colon and the trailing dollar sign. After substitution, the width of the fields is completely unchanged—the short values for `Rev` and `Author` are padded with spaces, and the long `Date` field is truncated by a hash character:

```
$Rev:: 13            $:  Revision of last commit
$Author:: harry      $:  Author of last commit
$Date:: 2006-03-15 0#$:  Date of last commit
```

The use of fixed-length keywords is especially handy when performing substitutions into complex file formats that themselves use fixed-length fields for data, or for which the stored size of a given data field is overbearingly difficult to modify from outside the format's native application (such as for Microsoft Office documents).

Warning

Be aware that because the width of a keyword field is measured in bytes, the potential for corruption of multi-byte values exists. For example, a username which contains some multi-byte UTF-8 characters might suffer truncation in the middle of the string of bytes which make up one of those characters. The result will be a mere truncation when viewed at the byte level, but will likely appear as a string with an incorrect or garbled final character when viewed as UTF-8 text. It is conceivable that certain applications, when asked to load the file, would notice the broken UTF-8 text and deem the entire file corrupt, refusing to operate on the file altogether.

svn:eol-style

Unless otherwise noted using a versioned file's `svn:mime-type` property, Subversion assumes the file contains human-readable data. Generally speaking, Subversion only uses this knowledge to determine if contextual difference reports for that file are possible. Otherwise, to Subversion, bytes are bytes.

This means that by default, Subversion doesn't pay any attention to the type of *end-of-line (EOL) markers* used in your files. Unfortunately, different operating systems use different tokens to represent the end of a line of text in a file. For example, the usual line ending token used by software on the Windows platform is a pair of ASCII control characters—carriage return (`CR`) and line feed (`LF`). Unix software, however, just uses the `LF` character to denote the end of a line.

Not all of the various tools on these operating systems are prepared to understand files that contain line endings in a format that differs from the *native line ending style* of the operating system on which they are running. Common results are that Unix programs treat the `CR` character present in Windows files as a regular character (usually rendered as `^M`), and that Windows programs combine all of the lines of a Unix file into one giant line because no carriage return-linefeed (or `CRLF`) character combination was found to denote the end of line.

This sensitivity to foreign EOL markers can become frustrating for folks who share a file across different operating systems. For example, consider a source code file, and developers that edit this file on both Windows and Unix systems. If all the developers always use tools which preserve the line ending style of the file, no problems occur.

But in practice, many common tools either fail to properly read a file with foreign EOL markers, or they convert the file's line endings to the native style when the file is saved. If the former is true for a developer, he has to use an external conversion utility (such as **dos2unix** or its companion, **unix2dos**) to prepare the file for editing. The latter case requires no extra preparation. But both cases result in a file that differs from the original quite literally on every line! Prior to committing his changes, the user has two choices. Either he can use a conversion utility to restore the modified file to the same line ending style that it was in before his edits were made. Or, he can simply commit the file—new EOL markers and all.

The result of scenarios like these include wasted time and unnecessary modifications to committed files. Wasted time is painful enough. But when commits change every line in a file, this complicates the job of determining which of those lines were changed in a non-trivial way. Where was that bug really fixed? On what line was a syntax error introduced?

The solution to this problem is the `svn:eol-style` property. When this property is set to a valid value, Subversion uses it to determine what special processing to perform on the file so that the file's line ending style isn't flip-flopping with every commit that comes from a different operating system. The valid values are:

`native`
This causes the file to contain the EOL markers that are native to the operating system on which Subversion was run. In other words, if a user on a Windows machine checks out a working copy that contains a file with an `svn:eol-style` property set to `native`, that file will contain `CRLF` EOL markers. A Unix user checking out a working copy which contains the same file will see `LF` EOL markers in his copy of the file.

Note that Subversion will actually store the file in the repository using normalized `LF` EOL markers regardless of the operating system. This is basically transparent to the user, though.

`CRLF`
This causes the file to contain `CRLF` sequences for EOL markers, regardless of the operating system in use.

`LF`
This causes the file to contain `LF` characters for EOL markers, regardless of the operating system in use.

`CR`
This causes the file to contain `CR` characters for EOL markers, regardless of the operating system in use. This line ending style is not very common. It was used on older Macintosh platforms (on which Subversion doesn't even run).

svn:externals

The `svn:externals` property contains instructions for Subversion to populate a versioned directory with one or more other checked-out Subversion working copies. For more information on this keyword and its use, see the section called “Externals Definitions”.

`svn:special`

The `svn:special` property is the only `svn:` property that isn't meant to be directly set or modified by users. Subversion automatically sets this property whenever a “special” object is scheduled for addition, such as a symbolic link. The repository stores an `svn:special` object as an ordinary file. However, when a client sees this property during checkouts or updates, it interprets the contents of the file and translates the item back into the special type of object. In versions of Subversion current at the time of writing, only versioned symbolic links have this property attached, but in future versions of Subversion other special types of nodes will probably use this property as well.

Note: Windows clients don't have symbolic links, and thus ignore any `svn:special` files coming from a repository that claim to be symbolic links. On Windows, the user ends up with an ordinary versioned file in the working copy.

`svn:needs-lock`

This property is used to signify that the file it's attached to ought to be locked before editing. The value of the property is irrelevant; Subversion will normalize its value to `*`. When present, the file will be read-only *unless* the user has explicitly locked the file. When a lock-token is present (as a result of running **svn lock**), the file becomes read-write. When the lock is released, the file becomes read-only again.

Necessary, sometimes one has to use the “lock-modify-unlock” model instead of Subversion's standard concurrent model. When a file consists of binary data, it's often difficult or impossible to merge two sets of changes made in parallel by different users. For this reason, Subversion 1.2 and later offers a feature known as *locking*, often known as “reserved checkouts” in other version control systems.

Subversion's locking feature has two main goals:

- *Serializing access to a resource*. Allow a user to grab an exclusive right to change to a file in the repository. If Harry reserves the right to change `foo.jpg`, then Sally should not be able to commit a change to it.
- *Aiding communication*. Prevent users from wasting time on unmergeable changes. If Harry has reserved the right to change `foo.jpg`, then it should be easy for Sally to notice this fact and avoid working on the file.

Subversion's locking feature is currently limited to files only—it's not yet possible to reserve access to a whole directory tree.

Three meanings of “lock”

In this section, and almost everywhere in this book, the words “lock” and “locking” describe a mechanism for mutual exclusion between users to avoid clashing commits. Unfortunately, there are two other sorts of “lock” with which Subversion, and therefore this book, sometimes needs to be concerned.

- *Working copy locks*, used internally by Subversion to prevent clashes between multiple Subversion clients operating on the same working copy. This is the sort of lock indicated by an `L` in the third column of **svn status** output, and removed by the **svn cleanup** command, as described in the section called “**svn cleanup**”.
- *Database locks*, used internally by the Berkeley DB backend to prevent clashes between multiple programs trying to access the database. This is the sort of lock whose unwanted persistence after an error can cause a repository to be “wedged”, as described in the section called “Repository Recovery”.

You can generally forget about these other sorts of lock, until something goes wrong that requires you to care about them. In this book, “lock” means the first sort unless the contrary is either clear from context or explicitly stated.

Creating locks

In the Subversion repository, a *lock* is a piece of metadata which grants exclusive access to one user to change a file. This user is said to be the *lock owner*. Each lock also has a unique identifier, typically a long string of characters, known as the *lock token*. The repository manages locks in a separate table, and enforces locks during a commit operation. If any commit transaction attempts to modify or delete the file (or delete a parent of the file), the repository will demand two pieces of information:

1. **User authentication**. The client performing the commit must be authenticated as the lock owner.
2. **Software authorization**. The user's working copy must send the lock token with the commit, proving that it knows exactly which lock it's using.

An example is in order, to demonstrate. Let's say that Harry has decided to change a JPEG image. To prevent other people from committing changes to the file, he locks the file in the repository using the **svn lock** command:

```
$ svn lock banana.jpg --message "Editing file for tomorrow's release."
'banana.jpg' locked by user 'harry'.

$ svn status
     K banana.jpg

$ svn info banana.jpg
Path: banana.jpg
Name: banana.jpg
URL: http://svn.example.com/repos/project/banana.jpg
Repository UUID: edb2f264-5ef2-0310-a47a-87b0ce17a8ec
Revision: 2198
Node Kind: file
Schedule: normal
Last Changed Author: frank
Last Changed Rev: 1950
```

```
$ svn unlock banana.c
'banana.c' unlocked.
```

Discovering locks

When a commit fails due to someone else's locks, it's fairly easy to learn about them. The easiest of these is **svn status --show-updates**:

```
$ whoami
sally

$ svn status --show-updates
M              23   bar.c
M    O         32   raisin.jpg
       *       72   foo.h
Status against revision:     105
```

In this example, Sally can see not only that her copy of `foo.h` is out-of-date, but that one of the two modified files she plans to commit is locked in the repository. The `O` symbol stands for “Other”, meaning that a lock exists on the file, and was created by somebody else. If she were to attempt a commit, the lock on `raisin.jpg` would prevent it. Sally is left wondering who made the lock, when, and why. Once again, **svn info** has the answers:

```
$ svn info http://svn.example.com/repos/project/raisin.jpg
Path: raisin.jpg
Name: raisin.jpg
URL: http://svn.example.com/repos/project/raisin.jpg
Repository UUID: edb2f264-5ef2-0310-a47a-87b0ce17a8ec
Revision: 105
Node Kind: file
Last Changed Author: sally
Last Changed Rev: 32
Last Changed Date: 2005-01-25 12:43:04 -0600 (Tue, 25 Jan 2005)
Lock Token: opaquelocktoken:fc2b4dee-98f9-0310-abf3-653ff3226e6b
Lock Owner: harry
Lock Created: 2005-02-16 13:29:18 -0500 (Wed, 16 Feb 2005)
Lock Comment (1 line):
Need to make a quick tweak to this image.
```

Just as **svn info** can be used to examine objects in the working copy, it can also be used to examine objects in the repository. If the main argument to **svn info** is a working copy path, then all of the working copy's cached information is displayed; any mention of a lock means that the working copy is holding a lock token (if a file is locked by another user or in another working copy, **svn info** on a working copy path will show no lock information at all). If the main argument to **svn info** is a URL, then the information reflects the latest version of an object in the repository; any mention of a lock describes the current lock on the object.

So in this particular example, Sally can see that Harry locked the file on February 16th to “make a quick tweak”. It being June, she suspects that he probably forgot all about the lock. She might phone Harry to complain and ask him to release the lock. If he's unavailable, she might try to forcibly break the lock herself or ask an administrator to do so.

Breaking and stealing locks

Of course, simply breaking a lock may not be enough. In the running example, Sally may not only want to break Harry's long-forgotten lock, but re-lock the file for her own use. She can accomplish this by running **svn unlock --force** and then **svn lock** back-to-back, but there's a small chance that somebody else might lock the file between the two commands. The simpler thing to is *steal* the lock, which in-

To learn more about repository hooks, see the section called “Hook Scripts”.

Lock Communication

We've seen how **svn lock** and **svn unlock** can be used to create, release, break, and steal locks. This satisfies the goal of serializing commit access to a file. But what about the larger problem of preventing wasted time?

For example, suppose Harry locks an image file and then begins editing it. Meanwhile, miles away, Sally wants to do the same thing. She doesn't think to run **svn status --show-updates**, so she has no idea that Harry has already locked the file. She spends hours editing the file, and when she tries to commit her change, she discovers that either the file is locked or that she's out-of-date. Regardless, her changes aren't mergeable with Harry's. One of these two people has to throw away their work, and a lot of time has been wasted.

Subversion's solution to this problem is provide a mechanism to remind users that a file ought to be locked *before* the editing begins.

The mechanism is a special property,

allowing users to edit and save the file anyway. Unfortunately, there's not much Subversion can do about this.

Peg and Operative Revisions

We make use of the ability to copy, move, rename, and completely replace files and directories on our computers all that time. And your version control system shouldn't get in the way of your doing these things with your version controlled files and directories, either. Subversion's file management support is quite liberating, affording almost as much flexibility for versioned files that you'd expect when manipulating your unversioned ones. But that flexibility means that across the lifetime of your repository, a given versioned resource might have many paths, and a given path might represent several entirely different versioned resources. And this introduces a certain level of complexity to your interactions with those paths and resources.

Subversion is pretty smart about noticing when an object's version history includes such “changes of address”. For example, if you ask for all the logs of a particular file that was renamed last week, Subversion happily provides all those logs—the revision in which the rename itself happened, plus the logs of relevant revisions both before and after that rename. So, most of the time, you don't even have to think about such things. But occasionally, Subversion needs your help to clear up ambiguities.

The simplest example of this occurs when a directory or file is deleted from version control, and then a new directory or file is created with the same name and added to version control. Clearly the thing you deleted and the thing you later added aren't the same thing, they merely happen to have had the same path, which we'll call `/trunk/object`. What, then, does it mean to ask Subversion about the history of `/trunk/object`? Are you asking about the thing currently at that location, or the old thing you deleted from that location? Are you asking about the operations that have happened to all the objects that have lived at that path? Clearly, Subversion needs a hint about what you are really asking.

And thanks to moves, versioned resource history can get far more twisted than that, even. For example, you might have a directory named `concept`, containing some nascent software project you've been toying with. Eventually, though, that project matures to the point that the idea seems to actually have some wings, so you do the unthinkable and decide to give the project a name. [9] Let's say you called your software Frabnaggilywort. At this point, it makes sense to rename the directory to reflect the project's new name, so `concept` is renamed to `frabnaggilywort`. Life goes on, Frabnaggilywort releases a 1.0 version, and is downloaded and used daily by hordes of people aiming to improve their lives.

It's a nice story, really, but it doesn't end there. Entrepreneur that you are, you've already got another think in the tank. So you make a new directory, `concept`, and the cycle begins again. In fact, the cycle begins again many times over the years, each time starting with that old `concept` directory, then sometimes seeing that directory renamed as the idea cures, sometimes seeing it deleted when you scrap the idea. Or, to get really sick, maybe you rename `concept` to something else for a while, but later rename the thing back to `concept` for some reason.

When scenarios like these occur, attempting to instruct Subversion to work with these re-used paths can be a little like instructing a motorist in Chicago's West Suburbs to drive east down Roosevelt Road and turn left onto Main Street. In a mere twenty minutes, you can cross “Main Street” in Wheaton, Glen Ellyn, and Lombard. And no, they aren't the same street. Our motorist—and our Subversion—need a little more detail in order to do the right thing.

In version 1.1, Subversion introduced a way for you to tell it exactly which Main Street you meant. It's called the *peg revision*, and it is a revision provided to Subversion for the sole purpose of identifying a unique line of history. Because at most one versioned resource may occupy a path at any given time—or, more precisely, in any one revision—the combination of a path and a peg revision is all that is needed to refer to a specific line of history. Peg revisions are specified to the Subversion command-line client using *at syntax*, so called because the syntax involves appending an “at sign” (`@`) and the peg revi-

[9] “You're not supposed to name it. Once you name it, you start getting attached to it.” — Mike Wazowski

sion to the end of the path with which the revision is associated.

But what of the `--revision (-r)` of which we've spoken so much in this book? That revision (or set of revisions) is called the *operative revision* (or *operative revision range*). Once a particular line of history has been identified using a path and peg revision, Subversion performs the requested operation using the operative revision(s). To map this to our Chicagoland streets analogy, if we are told to go to 606 N. Main Street in Wheaton, [10] we can think of “Main Street” as our path and “Wheaton” as our peg revision. These two pieces of information identify a unique path which can travelled (north or south on Main Street), and will keep us from travelling up and down the wrong Main Street in search of our destination. Now we throw in “606 N.” as our operative revision, of sorts, and we know *exactly* where to go.

The "peg-revision" algorithm

When the commandline client sees a command of the form:

```
$ svn command -r OPERATIVE-REV item@PEG-REV
```

...it performs the following algorithm:

- Go to revision *`PEG-REV`*, and find *`item`*. This locates a unique object in the repository.
- Trace the object's history backwards (through any possible renames) to its ancestor in revision *`OPERATIVE-REV`*.
- Perform the requested action on that ancestor, wherever it is located, or whatever its name might be.

[10] 606 N. Main Street, Wheaton, Illinois, is the home of the Wheaton History Center. Get it—“History Center”? It seemed appropriate….

```
$ svn cat -r 1 concept/IDEA@BASE
subversion/libsvn_client/ra.c:775: (apr_err=20014)
svn: Unable to find repository location for 'concept/IDEA' in revision 1
```

And when executed, it has the expected results. Peg revisions generally default to a value of `BASE` (the revision currently present in the working copy) when applied to working copy paths, and of `HEAD` when applied to URLs.

Let's ask the other question, then—in revision 1, what were the contents of whatever file occupied the address `concepts/IDEA` at the time? We'll use an explicit peg revision to help us out.

```
$ svn cat concept/IDEA@1
The idea behind this project is to come up with a piece of software
that can frab a naggily wort.  Frabbing naggily worts is tricky
business, and doing it incorrectly can have serious ramifications, so
we need to employ over-the-top input validation and data verification
mechanisms.
```

This appears to be the right output. The text even mentions frabbing naggily worts, so this is almost certainly the file which describes the software now called Frabnaggilywort. In fact, we can verify this using the combination of an explicit peg revision and explicit operative revision. We know that in `HEAD`, the Frabnaggilywort project is located in the `frabnaggilywort` directory. So we specify that we want to see how the line of history identified in `HEAD` as the path `frabnaggilywort/IDEA` looked in revision 1.

```
$ svn cat -r 1 frabnaggilywort/IDEA@HEAD
The idea behind this project is to come up with a piece of software
that can frab a naggily wort.  Frabbing naggily worts is tricky
business, and doing it incorrectly can have serious ramifications, so
we need to employ over-the-top input validation and data verification
mechanisms.
```

And the peg and operative revisions need not be so trivial, either. For example, say `frabnaggilywort` had been deleted from `HEAD`, but we know it existed in revision 20, and we want to see the diffs for its `IDEA` file between revisions 4 and 10. We can use the peg revision 20 in conjunction with the URL that would have held Frabnaggilywort's `IDEA` file in revision 20, and then use 4 and 10 as our operative revision range.

```
$ svn diff -r 4:10 http://svn.red-bean.com/projects/frabnaggilywort/IDEA@20
Index: frabnaggilywort/IDEA
===================================================================
--- frabnaggilywort/IDEA (revision 4)
+++ frabnaggilywort/IDEA (revision 10)
@@ -1,5 +1,5 @@
-The idea behind this project is to come up with a piece of software
-that can frab a naggily wort.  Frabbing naggily worts is tricky
-business, and doing it incorrectly can have serious ramifications, so
-we need to employ over-the-top input validation and data verification
-mechanisms.
+The idea behind this project is to come up with a piece of
+client-server software that can remotely frab a naggily wort.
+Frabbing naggily worts is tricky business, and doing it incorrectly
+can have serious ramifications, so we need to employ over-the-top
+input validation and data verification mechanisms.
```

Fortunately, most folks aren't faced with such complex situations. But when you are, remember that peg revisions are that extra hint Subversion needs to clear up ambiguity.

Externals Definitions

Sometimes it is useful to construct a working copy that is made out of a number of different checkouts. For example, you may want different subdirectories to come from different locations in a repository, or perhaps from different repositories altogether. You could certainly setup such a scenario by hand—using **svn checkout** to create the sort of nested working copy structure you are trying to achieve. But if this layout is important for everyone who uses your repository, every other user will need to perform the same checkout operations that you did.

Fortunately, Subversion provides support for *externals definitions*. An externals definition is a mapping of a local directory to the URL—and possibly a particular revision—of a versioned resource. In Subversion, you declare externals definitions in groups using the `svn:externals` property. You can create or modify this property using **svn propset** or **svn propedit** (see the section called "Why Properties?"). It can be set on any versioned directory, and its value is a multi-line table of subdirectories (relative to the versioned directory on which the property is set) and fully qualified, absolute Subversion repository URLs.

```
$ svn propget svn:externals calc
third-party/sounds             http://sounds.red-bean.com/repos
third-party/skins              http://skins.red-bean.com/repositories/skinproj
third-party/skins/toolkit -r21 http://svn.red-bean.com/repos/skin-maker
```

The convenience of the `svn:externals` property is that once it is set on a versioned directory, everyone who checks out a working copy with that directory also gets the benefit of the externals definition. In other words, once one person has made the effort to define those nested working copy checkouts, no one else has to bother—Subversion will, upon checkout of the original working copy, also checkout the external working copies.

The same thing will happen when others update their working copies and receive your changes to the externals definition.

The **svn status** command also recognizes externals definitions, displaying a status code of `X` for the disjoint subdirectories into which externals are checked out, and then recursing into those subdirectories to display the status of the external items themselves.

Tip

You should strongly consider using explicit revision numbers in all of your externals definitions. Doing so means that you get to decide when to pull down a different snapshot of external information, and exactly which snapshot to pull. Besides the common sense aspect of not being surprised by changes to third-party repositories that you might not have any control over, using explicit revision numbers also means that as you backdate your working copy to a previous revision, your externals definitions will also revert to the way they looked in that previous revision, which in turn means that the external working copies will be updated to match they way *they* looked back when your repository was at that previous revision. For software projects, this could be the difference between a successful and a failed build of an older snapshot of your complex codebase.

The support that exists for externals definitions in Subversion today can be a little misleading, though. First, an externals definition can only point to directories, not files. Second, the externals definition cannot point to relative paths (paths like `../../skins/myskin`). Third, the working copies created via the externals definition support are still disconnected from the primary working copy (on whose versioned directories the `svn:externals` property was actually set). And Subversion still only truly operates on non-disjoint working copies. So, for example, if you want to commit changes that you've made in one or more of those external working copies, you must run **svn commit** explicitly on those working copies—committing on the primary working copy will not recurse into any external ones.

Also, since the definitions themselves use absolute URLs, moving or copying a directory to which they are attached will not affect what gets checked out as an external (though the relative local target subdirectory will, of course, move with renamed directory). This can be confusing—even frustrating—in certain situations. For example, if you use externals definitions on a directory in your `/trunk` development line which point to other areas of that same line, and then you use **svn copy** to branch that line to some new location `/branches/my-branch`, the externals definitions on items in your new branch will still refer to versioned resources in `/trunk`. Be aware, too, that if you need to re-parent your working copy (using **svn switch --relocate**), externals definitions will *not* also be re-parented.

Finally, there might be times when you would prefer that **svn** subcommands would not recognize or otherwise operate on the external working copies created as the result of externals definition handling. In those instances, you can pass the `--ignore-externals` option to the subcommand.

Vendor branches

As is especially the case when developing software, the data that you maintain under version control is often closely related to, or perhaps dependent upon, someone else's data. Generally, the needs of your project will dictate that you stay as up-to-date as possible with the data provided by that external entity without sacrificing the stability of your own project. This scenario plays itself out all the time—anywhere that the information generated by one group of people has a direct effect on that which is generated by another group.

For example, software developers might be working on an application which makes use of a third-party library. Subversion has just such a relationship with the Apache Portable Runtime library (see the section called “The Apache Portable Runtime Library”). The Subversion source code depends on the APR library for all its portability needs. In earlier stages of Subversion's development, the project closely tracked APR's changing API, always sticking to the “bleeding edge” of the library's code churn. Now that both APR and Subversion have matured, Subversion attempts to synchronize with APR's library

API only at well-tested, stable release points.

Now, if your project depends on someone else's information, there are several ways that you could attempt to synchronize that information with your own. Most painfully, you could issue oral or written instructions to all the contributors of your project, telling them to make sure that they have the specific versions of that third-party information that your project needs. If the third-party information is maintained in a Subversion repository, you could also use Subversion's externals definitions to effectively “pin down” specific versions of that information to some location in your own working copy directory (see the section called “Externals Definitions”).

But sometimes you want to maintain custom modifications to third-party data in your own version control system. Returning to the software development example, programmers might need to make modifications to that third-party library for their own purposes. These modifications might include new functionality or bug fixes, maintained internally only until they become part of an official release of the third-party library. Or the changes might never be relayed back to the library maintainers, existing solely as custom tweaks to make the library further suit the needs of the software developers.

Now you face an interesting situation. Your project could house its custom modifications to the third-party data in some disjointed fashion, such as using patch files or full-fledged alternate versions of files and directories. But these quickly become maintenance headaches, requiring some mechanism by which to apply your custom changes to the third-party data, and necessitating regeneration of those changes with each successive version of the third-party data that you track.

The solution to this problem is to use *vendor branches*. A vendor branch is a directory tree in your own version control system that contains information provided by a third-party entity, or vendor. Each version of the vendor's data that you decide to absorb into your project is called a *vendor drop*.

Vendor branches provide two key benefits. First, by storing the currently supported vendor drop in your own version control system, the members of your project never need to question whether they have the right version of the vendor's data. They simply receive that correct version as part of their regular working copy updates. Secondly, because the data lives in your own Subversion repository, you can store your custom changes to it in-place—you have no more need of an automated (or worse, manual) method for swapping in your customizations.

General Vendor Branch Management Procedure

Managing vendor branches generally works like this. You create a top-level directory (such as `/vendor`) to hold the vendor branches. Then you import the third party code into a subdirectory of that top-level directory. You then copy that subdirectory into your main development branch (for example, `/trunk`) at the appropriate location. You always make your local changes in the main development branch. 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.

Perhaps an example will help to clarify this algorithm. We'll use a scenario where your development team is creating a calculator program that links against a third-party complex number arithmetic library, libcomplex. We'll begin with the initial creation of the vendor branch, and the import of the first vendor drop. We'll call our vendor branch directory `libcomplexctory` vendor/libcomplex/current. Now, we tag that version (see the section called “Tags”) and then copy it into the main development branch. Our copy will create a new directory called libcomplex in our existing calc project directory. It is in this copied version of the vendor data that we will make our customizations.

```
$ svn copy http://svn.example.com/repos/vendor/libcomplex/current  \
           http://svn.example.com/repos/vendor/libcomplex/1.0      \
           -m 'tagging libcomplex-1.0'
…
$ svn copy http://svn.example.com/repos/vendor/libcomplex/1.0  \
           http://svn.example.com/repos/calc/libcomplex        \
           -m 'bringing libcomplex-1.0 into the main branch'
…
```

We check out our project's main branch—which now includes a copy of the first vendor drop—and we get to work customizing the libcomplex code. Before we know it, our modified version of libcomplex is now completely integrated into our calculator program. [11]

A few weeks later, the developers of libcomplex release a new version of their library—version 1.1—which contains some features and functionality that we really want. We'd like to upgrade to this new version, but without losing the customizations we made to the existing version. What we essentially would like to do is to replace our current baseline version of libcomplex 1.0 with a copy of libcomplex 1.1, and then re-apply the custom modifications we previously made to that library to the new version. But we actually approach the problem from the other direction, applying the changes made to libcomplex between versions 1.0 and 1.1 to our modified copy of it.

To perform this upgrade, we checkout a copy of our vendor branch, and replace the code in the current directory with the new libcomplex 1.1 source code. We quite literally copy new files on top of existing files, perhaps exploding the libcomplex 1.1 release tarball atop our existing files and directories. The goal here is to make our current directory contain only the libcomplex 1.1 code, and to ensure that all that code is under version control. Oh, and we want to do this with as little version control history disturbance as possible.

After replacing the 1.0 code with 1.1 code, **svn status** will show files with local modifications as well as, perhaps, some unversioned or missing files. If we did what we were supposed to do, the unversioned files are only those new files introduced in the 1.1 release of libcomplex—we run **svn add** on those to get them under version control. The missing files are files that were in 1.0 but not in 1.1, and on those paths we run **svn delete**. Finally, once our current working copy contains only the libcomplex 1.1 code, we commit the changes we made to get it looking that way.

Our current branch now contains the new vendor drop. We tag the new version (in the same way we previously tagged the version 1.0 vendor drop), and then merge the differences between the tag of the previous version and the new current version into our main development branch.

```
$ cd working-copies/calc
$ svn merge http://svn.example.com/repos/vendor/libcomplex/1.0      \
            http://svn.example.com/repos/vendor/libcomplex/current  \
            libcomplex
… # resolve all the conflicts between their changes and our changes
$ svn commit -m 'merging libcomplex-1.1 into the main branch'
…
```

In the trivial use case, the new version of our third-party tool would look, from a files-and-directories point of view, just like the previous version. None of the libcomplex source files would have been deleted, renamed or moved to different locations—the new version would contain only textual modifica-

[11] And entirely bug-free, of course!

tions against the previous one. In a perfect world, our modifications would apply cleanly to the new version of the library, with absolutely no complications or conflicts.

But things aren't always that simple, and in fact it is quite common for source files to get moved around between releases of software. This complicates the process of ensuring that our modifications are still valid for the new version of code, and can quickly degrade into a situation where we have to manually recreate our customizations in the new version. Once Subversion knows about the history of a given source file—including all its previous locations—the process of merging in the new version of the library is pretty simple. But we are responsible for telling Subversion how the source file layout changed from vendor drop to vendor drop.

svn_load_dirs.pl

Vendor drops that contain more than a few deletes, additions and moves complicate the process of upgrading to each successive version of the third-party data. So Subversion supplies the **svn_load_dirs.pl** script to assist with this process. This script automates the importing steps we mentioned in the general vendor branch management procedure to make sure that mistakes are minimized. You will still be responsible for using the merge commands to merge the new versions of the third-party data into your main development branch, but **svn_load_dirs.pl** can help you more quickly and easily arrive at that stage.

In short, **svn_load_dirs.pl** is an enhancement to **svn import** that has several important characteristics:

- It can be run at any point in time to bring an existing directory in the repository to exactly match an external directory, performing all the necessary adds and deletes, and optionally performing moves, too.
- It takes care of complicated series of operations between which Subversion requires an intermediate commit—such as before renaming a file or directory twice.
- It will optionally tag the newly imported directory.
- It will optionally add arbitrary properties to files and directories that match a regular expression.

svn_load_dirs.pl takes three mandatory arguments. The first argument is the URL to the base Subversion directory to work in. This argument is followed by the URL—relative to the first argument—into which the current vendor drop will be imported. Finally, the third argument is the local directory to import. Using our previous example, a typical run of **svn_load_dirs.pl** might look like:

```
$ svn_load_dirs.pl http://svn.example.com/repos/vendor/libcomplex \
                   current                                        \
                   /path/to/libcomplex-1.1
…
```

You can indicate that you'd like **svn_load_dirs.pl** to tag the new vendor drop by passing the `-t` command-line option and specifying a tag name. This tag is another URL relative to the first program argument.

```
$ svn_load_dirs.pl -t libcomplex-1.1                              \
                   http://svn.example.com/repos/vendor/libcomplex \
                   current                                        \
                   /path/to/libcomplex-1.1
…
```

When you run **svn_load_dirs.pl**, it examines the contents of your existing “current” vendor drop, and compares them with the proposed new vendor drop. In the trivial case, there will be no files that are in

```
LC_TIME="C"
LC_ALL="C"
```

The output is a list of locale-related environment variables and their current values. In this example, the variables are all set to the default `C` locale, but users can set these variables to specific country/language code combinations. For example, if one were to set the `LC_TIME` variable to `fr_CA`, then programs would know to present time and date information formatted according a French-speaking Canadian's expectations. And if one were to set the `LC_MESSAGES` variable to `zh_TW`, then programs would know to present human-readable messages in Traditional Chinese. Setting the `LC_ALL` variable has the effect of changing every locale variable to the same value. The value of `LANG` is used as a default value for any locale variable that is unset. To see the list of available locales on a Unix system, run the command **locale -a**.

On Windows, locale configuration is done via the "Regional and Language Options" control panel item. There you can view and select the values of individual settings from the available locales, and even customize (at a sickening level of detail) several of the display formatting conventions.

Subversion's use of locales

The Subversion client, **svn**, honors the current locale configuration in two ways. First, it notices the value of the `LC_MESSAGES` variable and attempts to print all messages in the specified language. For example:

```
$ export LC_MESSAGES=de_DE
$ svn help cat
cat: Gibt den Inhalt der angegebenen Dateien oder URLs aus.
Aufruf: cat ZIEL[@REV]...
…
```

This behavior works identically on both Unix and Windows systems. Note, though, that while your operating system might have support for a certain locale, the Subversion client still may not be able to speak the particular language. In order to produce localized messages, human volunteers must provide translations for each language. The translations are written using the GNU gettext package, which results in translation modules that end with the `.mo` filename extension. For example, the German translation file is named `de.mo`. These translation files are installed somewhere on your system. On Unix, they typically live in `/usr/share/locale/`, while on Windows they're often found in the `\share\locale\` folder in Subversion's installation area. Once installed, a module is named after the program it provides translations for. For example, the `de.mo` file may ultimately end up installed as `/usr/share/locale/de/LC_MESSAGES/subversion.mo`. By browsing the installed `.mo` files, you can see which languages the Subversion client is able to speak.

The second way in which the locale is honored involves how **svn** interprets your input. The repository stores all paths, filenames, and log messages in Unicode, encoded as UTF-8. In that sense, the repository is *internationalized*—that is, the repository is ready to accept input in any human language. This means, however, that the Subversion client is responsible for sending only UTF-8 filenames and log messages into the repository. In order to do this, it must convert the data from the native locale into UTF-8.

For example, suppose you create a file named`caffè.txt`, and then when committing the file, you write the log message as "Adesso il caffè è più forte". Both the filename and log message contain non-ASCII characters, but because your locale is set to `it_IT`, the Subversion client knows to interpret them as Italian. It uses an Italian character set to convert the data to UTF-8 before sending them off to the repository.

Note that while the repository demands UTF-8 filenames and log messages, it *does not* pay attention to file contents. Subversion treats file contents as opaque strings of bytes, and neither client nor server makes an attempt to understand the character set or encoding of the contents.

Character set conversion errors

While using Subversion, you might get hit with an error related to character set conversions:

```
svn: Can't convert string from native encoding to 'UTF-8':
…
svn: Can't convert string from 'UTF-8' to native encoding:
…
```

Errors like this typically occur when the Subversion client has received a UTF-8 string from the repository, but not all of the characters in that string can be represented using the encoding of the current locale. For example, if your locale is `en_US` but a collaborator has committed a Japanese filename, you're likely to see this error when you receive the file during an **svn update**.

The solution is either to set your locale to something which *can* represent the incoming UTF-8 data, or to change the filename or log message in the repository. (And don't forget to slap your collaborator's hand—projects should decide on common languages ahead of time, so that all participants are using the same locale.)

Using External Differencing Tools

The presence of `--diff-cmd` and `--diff3-cmd` options, and similarly named runtime configuration parameters (see the section called "Config"), can lead to a false notion of how easy it is to use external differencing (or "diff") and merge tools with Subversion. While Subversion can use most of popular such tools available, the effort invested in setting this up often turns out to be non-trivial.

The interface between Subversion and external diff and merge tools harkens back to a time when Subversion's only contextual differencing capabilities were built around invocations of the GNU diffutils toolchain, specifically the **diff** and **diff3** utilities. To get the kind of behavior Subversion needed, it called these utilities with more than a handful of options and parameters, most of which were quite specific to the utilities. Some time later, Subversion grew its own internal differencing library, and as a failover mechanism, [12] the `--diff-cmd` and `--diff3-cmd` options were added to the Subversion command-line client so users could more easily indicate that they preferred to use the GNU diff and diff3 utilities instead of the newfangled internal diff library. If those options were used, Subversion would simply ignore the internal diff library, and fall back to running those external programs, lengthy argument lists and all. And that's where things remain today.

It didn't take long for folks to realize that having such easy configuration mechanisms for specifying that Subversion should use the external GNU diff and diff3 utilities located at a particular place on the system could be applied toward the use of other diff and merge tools, too. After all, Subversion didn't actually verify that the things it was being told to run were members of the GNU diffutils toolchain. But the only configurable aspect of using those external tools is their location on the system—not the option set, parameter order, etc. Subversion continues throwing all those GNU utility options at your external diff tool regardless of whether or not that program can understand those options. And that's where things get unintuitive for most users.

The key to using external diff and merge tools (other than GNU diff and diff3, of course) with Subversion is to use wrapper scripts which convert the input from Subversion into something that your differencing tool can understand, and then to convert the output of your tool back into a format which Subversion expects—the format that the GNU tools would have used. The following sections cover the specifics of those expectations.

[12]Subversion developers are good, but even the best make mistakes.

Note

The decision on when to fire off a contextual diff or merge as part of a larger Subversion operation is made entirely by Subversion, and is affected by, among other things, whether or not the files being operated on are human-readable as determined by their `svn:mime-type` property. This means, for example, that even if you had the niftiest Microsoft Word-aware differencing or merging tool in the Universe, it would never be invoked by Subversion so long as your versioned Word documents had a configured MIME type that denoted that they were not human-readable (such as `application/msword`). For more about MIME type settings, see the section called "`svn:mime-type`"

External diff

Subversion calls external diff programs with parameters suitable for the GNU diff utility, and expects only that the external program return with a successful error code. For most alternative diff program, only the sixth and seventh arguments, the paths of the files which represent the left and right sides of the diff, respectively, are of interest. Note that Subversion runs the diff program once per modified file covered by the Subversion operation, so if your program runs in an asynchronous fashion (or "backgrounded"), you might have several instances of it all running simultaneously. Finally, Subversion expects that your program return an errorcode of 0 if your program detected differences, or 1 if it did not—any other errorcode is considered a fatal error. [13]

[13]The GNU diff manual page puts it this way: "An exit status of 0 means no differences were found, 1 means some differences were found, and 2 means trouble."

```
SET DIFF="C:\Program Files\Funky Stuff\My Diff Tool.exe"

REM Subversion provides the paths we need as the sixth and seventh
REM parameters.
SET LEFT=%6
SET RIGHT=%7

REM Call the diff command (change the following line to make sense for
REM your merge program).
%DIFF% --left %LEFT% --right %RIGHT%

REM Return an errorcode of 0 if no differences were detected, 1 if some were.
REM Any other errorcode will be treated as fatal.
```

External diff3

Subversion calls external merge programs with parameters suitable for the GNU diff3 utility, expecting that the external program return with a successful error code and that the full file contents which result from the completed merge operation are printed on the standard output stream (so that Subversion can redirect them into the appropriate version controlled file). For most alternative merge programs, only the ninth, tenth, and eleventh arguments, the paths of the files which represent the “mine”, “older”, and “yours” inputs, respectively, are of interest. Note that because Subversion depends on the output of your merge program, you wrapper script must not exit before that output has been delivered to Subversion. When it finally does exit, it should return an errorcode of 0 if the merge was successful, or 1 if unresolved conflicts remain in the output—any other errorcode is considered a fatal error.

Example 7.4, “diff3wrap.sh” and Example 7.5, “diff3wrap.bat” are templates for external merge tool wrappers in the Bourne shell and Windows batch scripting languages, respectively.

Example 7.4. diff3wrap.sh

```
#!/bin/sh

# Configure your favorite diff3/merge program here.
DIFF3="/usr/local/bin/my-merge-tool"

# Subversion provides the paths we need as the ninth, tenth, and eleventh
# parameters.
MINE=${9}
OLDER=${10}
YOURS=${11}

# Call the merge command (change the following line to make sense for
# your merge program).
$DIFF3 --older $OLDER --mine $MINE --yours $YOURS

# After performing the merge, this script needs to print the contents
# of the merged file to stdout.  Do that in whatever way you see fit.
# Return an errorcode of 0 on successful merge, 1 if unresolved conflicts
# remain in the result.  Any other errorcode will be treated as fatal.
```

Example 7.5. diff3wrap.bat

pipe. Also, note that a URL uses ordinary slashes even though the native (non-URL) form of a path on Windows uses backslashes.

Finally, it should be noted that the Subversion client will automatically encode URLs as necessary, just like a web browser does. For example, if a URL contains a space or upper-ASCII character:

```
$ svn checkout "http://host/path with space/project/españa"
```

…then Subversion will escape the unsafe characters and behave as if you had typed:

```
$ svn checkout http://host/path%20with%20space/project/espa%C3%B1a
```

If the URL contains spaces, be sure to place it within quote marks, so that your shell treats the whole thing as a single argument to the **svn** program.

Chapter 8. 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 8.1, "A Brief Inventory of the Subversion Libraries". 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 8.1. A Brief Inventory of the Subversion Libraries

Library	Description
libsvn_client	Primary interface for client programs
libsvn_delta	Tree and byte-stream differencing routines
libsvn_diff	Contextual differencing and merging routines
libsvn_fs	Filesystem commons and module loader
libsvn_fs_base	The Berkeley DB filesystem back-end
libsvn_fs_fs	The native filesystem (FSFS) back-end
libsvn_ra	Repository Access commons and module loader
libsvn_ra_dav	The WebDAV Repository Access module
libsvn_ra_local	The local Repository Access module
libsvn_ra_svn	The custom protocol Repository Access module
libsvn_repos	Repository interface
libsvn_subr	Miscellaneous helpful subroutines
libsvn_wc	The working copy management library
mod_authz_svn	Apache authorization module for Subversion repositories access via WebDAV
mod_dav_svn	Apache module for mapping WebDAV operations to Subversion ones

The fact that the word "miscellaneous" only appears once in Table 8.1, "A Brief Inventory of the Subversion Libraries" 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.

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, libsvn_ra_local, and libsvn_ra_svn all implement the same interface. And all three communicate with the Repository Layer—libsvn_ra_dav and libsvn_ra_svn do so across a network, and libsvn_ra_local connects to it directly. The libsvn_fs_base and libsvn_fs_fs libraries are another example of this.

The client itself also highlights modularity in the Subversion design. While Subversion itself comes with only a command-line client program, there are several third party programs which provide various forms of client GUI. 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 one of two available abstract storage backends—either a Berkeley DB database environment, or a flat-file representation. (To learn more about the two repository back-ends, see the section called "Repository Data Stores".) However, there has been considerable interest by the development community in giving future releases of Subversion 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.

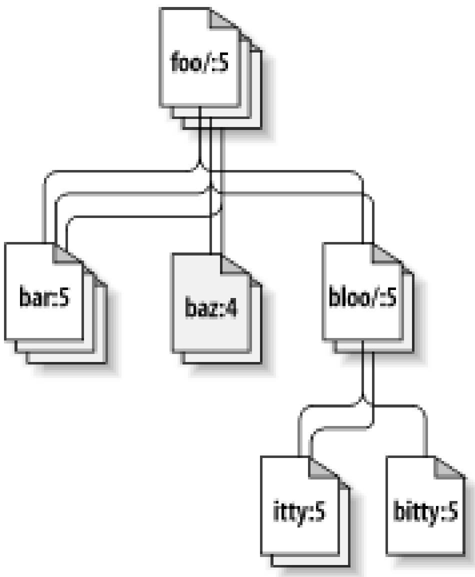
Database transactions generally encompass small operations related specifically to the modification of data in the database itself (such as changing the contents of a table row). Subversion transactions are larger in scope, encompassing higher-level operations like making modifications to a set of files 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`

Of course, the Subversion filesystem has a nifty third dimension that most filesystems do not have—Time! [1] 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 8.2, "Versioning time—the third dimension!" shows revision history as an added dimension to the Subversion filesystem universe.

Figure 8.2. Versioning time—the third dimension!



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

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

faces 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 8.1. 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, pool);
  if (! err)
    {
      /* No error?  Excellent!  Print a brief report of our success.  */
      printf ("Directory '%s' was successfully added as new revision "
              "'%ld'.\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 occurred 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 occurred while attempting to add directory '%s' "
                "to the repository at '%s'.\n",
                new_directory, repos_path);
      }

    /* Return the result of the attempted commit to our caller.  */
    return err;
  }
```

In the previous code segment, calls were made to both the repository and filesystem interfaces. 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. (For more information regarding Subversion's repository hooks, see the section called “Hook Scripts”.)

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_dav, libsvn_ra_local, and libsvn_ra_svn), 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 or libsvn_ra_svn's server, **svnserve**.

Since Subversion uses URLs to identify its repository resources, the protocol portion of the URL schema (usually `file:`, `http:`, `https:`, or `svn:`) 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**:

```
$ svn --version
svn, version 1.2.3 (r15833)
   compiled Sep 13 2005, 22:45:22

Copyright (C) 2000-2005 CollabNet.
Subversion is open source software, see http://subversion.tigris.org/
This product includes software developed by CollabNet (http://www.Collab.Net/).

The following repository access (RA) modules are available:

* ra_dav : Module for accessing a repository via WebDAV (DeltaV) protocol.
  - handles 'http' scheme
  - handles 'https' scheme
* ra_svn : Module for accessing a repository using the svn network protocol.
  - handles 'svn' scheme
* ra_local : Module for accessing a repository on local disk.
  - handles 'file' scheme
```

RA-DAV (Repository Access Using HTTP/DAV)

The libsvn_ra_dav library is designed for use by clients that are being run on different machines than the servers with which they communicating, specifically servers 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 primary network 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, system administrators typically don't even have to change their firewall configurations to allow Subversion to work.

Subversion uses HTTP and WebDAV (with DeltaV) to communicate with an Apache server. 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 and later versions come 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 one 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.

RA-SVN (Custom Protocol Repository Access)

In addition to the standard HTTP/WebDAV protocol, Subversion also provides an RA implementation that uses a custom protocol. The libsvn_ra_svn module implements its own network socket connectivity, and communicates with a stand-alone server—the `svnserve` program—on the machine that hosts the repository. Clients access the repository using the `svn://` schema.

This RA implementation lacks most of the advantages of Apache mentioned in the previous section; however, it may be appealing to some system administrators nonetheless. It is dramatically easier to configure and run; setting up an `svnserve` process is nearly instantaneous. It is also much smaller (in terms of lines of code) than Apache, making it much easier to audit, for security reasons or otherwise. Furthermore, some system administrators may already have an SSH security infrastructure in place, and want Subversion to use it. Clients using ra_svn can easily tunnel the protocol over SSH.

RA-Local (Direct Repository Access)

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.

Subversion requires that the server name included as part of the `file:` URL be either `localhost` or empty, and that there be no port specification. In other words, your URLs should look like either `file://localhost/path/to/repos` or `file:///path/to/repos`.

Also, be aware that Subversion's `file:` URLs cannot be used in a regular web browser the way typical `file:` URLs can. When you attempt to view a `file:` URL in a regular web browser, it reads and displays the contents of the file at that location by examining the filesystem directly. However, Subversion's resources exist in a virtual filesystem (see the section called “Repository Layer”), and your browser will not understand how to read that filesystem.

Your RA Library Here

For those who wish to access a Subversion repository using still another protocol, 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 library can use existing network protocols, or you can invent your own. You could use 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

[2] Subversion uses ANSI system calls and datatypes as much as possible.
[3]

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 or Perl script—Subversion has some support for this via the Simplified Wrapper and Interface Generator (SWIG). The SWIG bindings for Subversion are located in `subversion/bindings/swig` and whilst still maturing, they are in 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 type checking provided by these languages are often better at handling information that comes from users. As you know, humans are proficient at botching up input to a program, and scripting languages tend to 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 a sample program that uses Subversion's Python SWIG bindings to recursively crawl the youngest repository revision, and print the various paths reached during the crawl.

Example 8.2. Using the Repository Layer with Python

```
#!/usr/bin/python

"""Crawl a repository, printing versioned object path names."""

import sys
import os.path
import svn.fs, svn.core, svn.repos

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."""

    # Print the name of this path.
    print directory + "/"

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

    # Use an iteration subpool.
    subpool = svn.core.svn_pool_create(pool)

    # Loop over the entries.
    names = entries.keys()
    for name in names:
        # Clear the iteration subpool.
        svn.core.svn_pool_clear(subpool)
```

```
        # Calculate the entry's full path.
        full_path = directory + '/' + 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 svn.fs.svn_fs_is_dir(root, full_path, subpool):
            crawl_filesystem_dir(root, full_path, subpool)
        else:
            # Else it's a file, so print its path here.
            print full_path

    # Destroy the iteration subpool.
    svn.core.svn_pool_destroy(subpool)

def crawl_youngest(pool, repos_path):
    """Open the repository at REPOS_PATH, and recursively crawl its
    youngest revision."""

    # Open the repository at REPOS_PATH, and get a reference to its
    # versioning filesystem.
    repos_obj = svn.repos.svn_repos_open(repos_path, pool)
    fs_obj = svn.repos.svn_repos_fs(repos_obj)

    # Query the current youngest revision.
    youngest_rev = svn.fs.svn_fs_youngest_rev(fs_obj, pool)

    # Open a root object representing the youngest (HEAD) revision.
    root_obj = svn.fs.svn_fs_revision_root(fs_obj, youngest_rev, pool)

    # Do the recursive crawl.
    crawl_filesystem_dir(root_obj, "", pool)

if __name__ == "__main__":
    # Check for sane usage.
    if len(sys.argv) != 2:
        sys.stderr.write("Usage: %s REPOS_PATH\n"
                         % (os.path.basename(sys.argv[0])))
        sys.exit(1)

    # Canonicalize (enough for Subversion, at least) the repository path.
    repos_path = os.path.normpath(sys.argv[1])
    if repos_path == '.':
        repos_path = ''

    # Call the app-wrapper, which takes care of APR initialization/shutdown
    # and the creation and cleanup of our top-level memory pool.
    svn.core.run_app(crawl_youngest, repos_path)
```

This same program in C would need to deal with custom datatypes (such as those provided by the APR library) for representing the hash of entries and the list of paths, but Python has hashes (called “dictionaries”) and lists as built-in datatypes, and provides a rich collection of functions for operating on those types. So SWIG (with the help of some customizations in Subversion's language bindings layer) takes care of mapping those custom datatypes into the native datatypes of the target language. This provides a more intuitive interface for users of that language.

The Subversion Python bindings can be used for working copy operations, too. 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 to recreate a scaled-down version of the **svn status** command.

Example 8.3. A Python Status Crawler

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

"""Crawl a working copy directory, printing status information."""

import sys
import os.path
import getopt
import svn.core, svn.client, svn.wc

def generate_status_code(status):
    """Translate a status value into a single-character status code,
```

```
    # Do the status crawl, using _status_callback() as our callback function.
    svn.client.svn_client_status(wc_path, None, _status_callback,
                                 1, verbose, 0, 0, ctx, pool)

def usage_and_exit(errorcode):
    """Print usage message, and exit with ERRORCODE."""
```

- 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 directories. 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.

The following is an example of an actual entries file:

Example 8.4. Contents of a Typical `.svn/entries` File

```
<?xml version="1.0" encoding="utf-8"?>
<wc-entries
   xmlns="svn:">
<entry
   committed-rev="1"
   name=""
   committed-date="2005-04-04T13:32:28.526873Z"
   url="http://svn.red-bean.com/repos/greek-tree/A/D"
   last-author="jrandom"
   kind="dir"
   uuid="4e820d15-a807-0410-81d5-aa59edf69161"
   revision="1"/>
<entry
   name="lambda"
   copied="true"
   kind="file"
   copyfrom-rev="1"
   schedule="add"
   copyfrom-url="http://svn.red-bean.com/repos/greek-tree/A/B/lambda"/>
<entry
   committed-rev="1"
   name="gamma"
   text-time="2005-12-11T16:32:46.000000Z"
   committed-date="2005-04-04T13:32:28.526873Z"
   checksum="ada10d942b1964d359e048dbacff3460"
   last-author="jrandom"
   kind="file"
   prop-time="2005-12-11T16:32:45.000000Z"/>
<entry
   name="zeta"
   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 (called the “this directory” entry, and noted as having an empty value for its *`name`* attribute), 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

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

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. RFCs 2518 and 3253 describe the WebDAV/DeltaV extensions to HTTP, and are 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 Web Folders (which are just WebDAV-ready network locations) as if they were regular shared folders. Mac OS X also has this capability, as do the Nautilus and Konqueror browsers (under GNOME and KDE, respectively).

How does all of this apply to Subversion? The mod_dav_svn Apache module uses HTTP, extended by WebDAV and DeltaV, as one of its network protocols. Subversion uses mod_dav_svn to map between Subversion's versioning concepts and those of RFCs 2518 and 3253.

For a more thorough discussion of WebDAV, how it works, and how Subversion uses it, see Appendix B, *WebDAV and Autoversioning*. Among other things, that appendix discusses the degree to which Subversion adheres to the generic WebDAV specification, and how that affects interoperability with generic WebDAV clients.

Programming with Memory Pools

Almost every developer who has used the C programming language has at some point sighed at the daunting task of managing memory usage. Allocating enough memory to use, keeping track of those allocations, freeing the memory when you no longer need it—these tasks can be quite complex. And of course, failure to do those things properly can result in a program that crashes itself, or worse, crashes the computer. Fortunately, the APR library that Subversion depends on for portability provides the apr_pool_t type, which represents a pool from which the application may allocate 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 `malloc()` and friends, programs that link against APR can simply request that a pool of memory be created (using the `apr_pool_create()`

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 8.5. 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;

      /* Clear the per-iteration SUBPOOL.  */
      svn_pool_clear (subpool);

      /* 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 (hash_pool, child_path);

          /* Add the path to the array.  */
          (*((const char **) apr_array_push (files))) = child_path;
        }
    }

  /* Destroy SUBPOOL.  */
  svn_pool_destroy (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;

  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

[5]Note that the URL checked out in the example above ends not with `svn`, but with a subdirectory thereof called

Make and Test Your Changes

With the code and community policy understanding in hand, you are ready to make your changes. It is best to try to make smaller but related sets of changes, even tackling larger tasks in stages, instead of making huge, sweeping modifications. Your proposed changes will be easier to understand (and therefore easier to review) if you disturb the fewest lines of code possible to accomplish your task properly. After making each set of proposed changes, your Subversion tree should be in a state in which the software compiles with no warnings.

Subversion has a fairly thorough [7] 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 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

[7] You might want to grab some popcorn. “Thorough”, in this instance, translates to somewhere in the neighborhood of thirty minutes of non-interactive machine churn.

Chapter 9. Subversion Complete Reference

This chapter is intended to be a complete reference to using Subversion. This includes the command line client (**svn**) and all its subcommands, as well as the repository administration programs (**svnadmin** and **svnlook**) and their respective subcommands.

The Subversion Command Line Client: svn

To use the command line client, you type **svn**, the subcommand you wish to use [1], and any switches or targets that you wish to operate on—there is no specific order that the subcommand and the switches must appear in. For example, all of the following are valid ways to use **svn status**:

```
$ svn -v status
$ svn status -v
$ svn status -v myfile
```

You can find many more examples of how to use most client commands in Chapter 3, *Guided Tour* and commands for managing properties in the section called "Properties".

svn Switches

While Subversion has different switches for its subcommands, all switches are global—that is, each switch is guaranteed to mean the same thing regardless of the subcommand you use it with. For example, `--verbose` (`-v`) always means "verbose output", regardless of the subcommand you use it with.

`--auto-props`
Enables auto-props, overriding the `enable-auto-props` directive in the `config` file.

`--config-dir` *`DIR`*
Instructs Subversion to read configuration information from the specified directory instead of the default location (`.subversion` in the user's home directory).

`--diff-cmd` *`CMD`*

[1] Yes, yes, you don't need a subcommand to use the `--version` switch, but we'll get to that in just a minute.

the section called “Config” for ways to specify a default editor.

`--encoding` *`ENC`*

Tells Subversion that your commit message is encoded in the charset provided. The default is your operating system's native locale, and you should specify the encoding if your commit message is in any other encoding.

`--extensions` (`-x`) *`ARGS`*

Specifies an argument or arguments that Subversion should pass to an external diff command when providing differences between files. If you wish to pass multiple arguments, you must enclose all of them in quotes (for example, **svn diff --diff-cmd /usr/bin/diff -x "-b -E"**). This switch can *only* be used if you also pass the `--diff-cmd` switch.

`--file` (`-F`) *`FILENAME`*

Uses the contents of the file passed as an argument to this switch for the specified subcommand.

`--force`

Forces a particular command or operation to run. There are some operations that Subversion will prevent you from doing in normal usage, but you can pass the force switch to tell Subversion “I know what I'm doing as

`--no-auto-props`

Disables auto-props, overriding the `enable-auto-props` directive in the `config` file.

`--no-diff-added`

Prevents Subversion from printing differences for added files. The default behavior when you add a file is for **svn diff** to print the same differences that you would see if you had added the entire contents of an existing (empty) file.

`--no-diff-deleted`

```
$ svn log -r 1729:1744
$ svn log -r {2001-12-04}:{2002-02-17}
$ svn log -r 1729:{2002-02-17}
```

See the section called “Revision Keywords” for more information.

`--revprop`
Operates on a revision property instead of a Subversion property specific to a file or directory. This switch requires that you also pass a revision with the `--revision` (`-r`) switch. See the section called “Unversioned Properties” for more details on unversioned properties.

`--show-updates` (`-u`)
Causes the client to display information about which files in your working copy are out-of-date. This doesn't actually update any of your files—it just shows you which files will be updated if you run **svn update**.

`--stop-on-copy`
Causes a Subversion subcommand which is traversing the history of a versioned resource to stop harvesting that historical information when a copy—that is, a location in history where that resource was copied from another location in the repository—is encountered.

`--strict`
Causes Subversion to use strict semantics, a notion which is rather vague unless talking about specific subcommands.

`--targets` *`FILENAME`*
Tells Subversion to get the list of files that you wish to operate on from the filename you provide instead of listing all the files on the command line.

`--username` *`NAME`*
Indicates that you are providing your username for authentication on the command line—otherwise, if it is needed, Subversion will prompt you for it.

`--verbose` (`-v`)
Requests that the client print out as much information as it can while running any subcommand. This may result in Subversion printing out additional fields, detailed information about every file, or additional information regarding its actions.

`--version`
Prints the client version info. This information not only includes the version number of the client, but also a listing of all repository access modules that the client can use to access a Subversion repository.

`--xml`
Prints output in XML format.

svn Subcommands

Name

svn add -- Add files, directories, or symbolic links.

Synopsis

```
svn add PATH...
```

Description

Add files, directories, or symbolic links to your working copy and schedule them for addition to the repository. They will be uploaded and added to the repository on your next commit. If you add something and change your mind before committing, you can unschedule the addition using **svn revert**.

Alternate Names

None

Changes

Working Copy

Accesses Repository

No

Switches

```
--targets FILENAME
--non-recursive (-N)
--quiet (-q)
--config-dir DIR
--auto-props
--no-auto-props
--force
```

Examples

To add a file to your working copy:

```
$ svn add foo.c
A         foo.c
```

When adding a directory, the default behavior of **svn add** is to recurse:

```
$ svn add testdir
A         testdir
A         testdir/a
A         testdir/b
A         testdir/c
A         testdir/d
```

You can add a directory without adding its contents:

```
$ svn add --non-recursive otherdir
A         otherdir
```

Normally, the command **svn add *** will skip over any directories that are already under version control. Sometimes, however, you may want to add every unversioned object in your working copy, including those hiding deeper down. Passing the `--force` option makes **svn add** recurse into versioned directories:

```
$ svn add * --force
A         foo.c
A         somedir/bar.c
A         otherdir/docs/baz.doc
…
```

Name

svn blame -- Show author and revision information in-line for the specified files or URLs.

Synopsis

```
svn blame TARGET[@REV]...
```

Description

Show author and revision information in-line for the specified files or URLs. Each line of text is annotated at the beginning with the author (username) and the revision number for the last change to that line.

Alternate Names

praise, annotate, ann

Changes

Nothing

Accesses Repository

Yes

Switches

```
--revision (-r) REV
--username USER
--password PASS
--no-auth-cache
--non-interactive
--config-dir DIR
--verbose
```

Examples

If you want to see blame annotated source for `readme.txt` in your test repository:

```
$ svn blame http://svn.red-bean.com/repos/test/readme.txt
     3      sally This is a README file.
     5      harry You should read this.
```

Name

svn cat -- Output the contents of the specified files or URLs.

Synopsis

```
svn cat TARGET[@REV]...
```

Description

Output the contents of the specified files or URLs. For listing the contents of directories, see **svn list**.

Alternate Names

None

Changes

Nothing

Accesses Repository

Yes

Switches

```
--revision (-r) REV
--username USER
--password PASS
--no-auth-cache
--non-interactive
--config-dir DIR
```

Examples

If you want to view readme.txt in your repository without checking it out:

```
$ svn cat http://svn.red-bean.com/repos/test/readme.txt
This is a README file.
You should read this.
```

Tip

If your working copy is out of date (or you have local modifications) and you want to see the `HEAD` revision of a file in your working copy, **svn cat** will automatically fetch the `HEAD` revision when you give it a path:

```
$ cat foo.c
This file is in my local working copy
and has changes that I've made.
```

```
$ svn cat foo.c
Latest revision fresh from the repository!
```

Name

svn checkout -- Check out a working copy from a repository.

Synopsis

```
svn checkout URL[@REV]... [PATH]
```

Description

Check out a working copy from a repository. If *PATH* is omitted, the basename of the URL will be used as the destination. If multiple URLs are given each will be checked out into a subdirectory of *PATH*, with the name of the subdirectory being the basename of the URL.

Alternate Names

co

Changes

Creates a working copy.

Accesses Repository

Yes

Switches

```
--revision (-r) REV
--quiet (-q)
--non-recursive (-N)
--username USER
--password PASS
--no-auth-cache
--non-interactive
--ignore-externals
--config-dir DIR
```

Examples

Check out a working copy into a directory called `mine`:

```
$ svn checkout file:///tmp/repos/test mine
A  mine/a
A  mine/b
Checked out revision 2.
$ ls
mine
```

Check out 2 different directories into two separate working copies:

```
$ svn checkout file:///tmp/repos/test  file:///tmp/repos/quiz
```

```
A  test/a
A  test/b
Checked out revision 2.
A  quiz/l
A  quiz/m
Checked out revision 2.
$ ls
quiz  test
```

Check out 2 different directories into two separate working copies, but place both into a directory called `working-copies`:

```
$ svn checkout file:///tmp/repos/test  file:///tmp/repos/quiz working-copies
A  working-copies/test/a
A  working-copies/test/b
Checked out revision 2.
A  working-copies/quiz/l
A  working-copies/quiz/m
Checked out revision 2.
$ ls
working-copies
```

If you interrupt a checkout (or something else interrupts your checkout like loss of connectivity, etc.), you can restart it either by issuing the identical checkout command again, or by updating the incomplete working copy:

```
$ svn checkout file:///tmp/repos/test test
A  test/a
A  test/b
^C
svn: The operation was interrupted
svn: caught SIGINT

$ svn checkout file:///tmp/repos/test test
A  test/c
A  test/d
^C
svn: The operation was interrupted
svn: caught SIGINT

$ cd test
$ svn update
A  test/e
A  test/f
Updated to revision 3.
```

Name

svn cleanup -- Recursively clean up the working copy.

Synopsis

```
svn cleanup [PATH...]
```

Description

Recursively clean up the working copy, removing working copy locks and resuming unfinished operations. If you ever get a “working copy locked” error, run this command to remove stale locks and get your working copy into a usable state again.

If, for some reason, an **svn update** fails due to a problem running an external diff program (e.g. user input or network failure), pass the `--diff3-cmd` to allow cleanup to complete any merging with your external diff program. You can also specify any configuration directory with the `--config-dir` switch, but you should need these switches extremely infrequently.

Alternate Names

None

Changes

Working copy

Accesses Repository

No

Switches

```
--diff3-cmd CMD
--config-dir DIR
```

Examples

Well, there's not much to the examples here as **svn cleanup** generates no output. If you pass no *PATH*, “`.`” is used.

```
$ svn cleanup

$ svn cleanup /path/to/working-copy
```

Name

svn commit -- Send changes from your working copy to the repository.

Synopsis

```
svn commit [PATH...]
```

Description

Send changes from your working copy to the repository. If you do not supply a log message with your commit by using either the `--file` or `--message` switch, **svn** will launch your editor for you to compose a commit message. See the `editor-cmd` section in the section called “Config”.

svn commit will send found lock tokens and release locks on all *PATHS* committed (recursively) unless `--no-unlock` is passed.

Tip

If you begin a commit and Subversion launches your editor to compose the commit message, you can still abort without committing your changes. If you want to cancel your commit, just quit your editor without saving your commit message and Subversion will prompt you to either abort the commit, continue with no message, or edit the message again.

Alternate Names

ci (short for “check in”; not “co”, which is short for “checkout”)

Changes

Working copy, repository

Accesses Repository

Yes

Switches

```
--message (-m) TEXT
--file (-F) FILE
--quiet (-q)
--no-unlock
--non-recursive (-N)
--targets FILENAME
--force-log
--username USER
--password PASS
--no-auth-cache
--non-interactive
--encoding ENC
--config-dir DIR
```

Examples

Commit a simple modification to a file with the commit message on the command line and an implicit target of your current directory (“`.`”):

Name

svn copy -- Copy a file or directory in a working copy or in the repository.

Synopsis

```
svn copy SRC DST
```

Description

Copy a file in a working copy or in the repository. *SRC* and *DST* can each be either a working copy (WC) path or URL:

WC -> WC
Copy and schedule an item for addition (with history).

WC -> URL
Immediately commit a copy of WC to URL.

URL -> WC
Check out URL into WC, and schedule it for addition.

URL -> URL
Complete server-side copy. This is usually used to branch and tag.

Note

You can only copy files within a single repository. Subversion does not support cross-repository copying.

Alternate Names

cp

Changes

Repository if destination is a URL.

Working copy if destination is a WC path.

Accesses Repository

If source or destination is in the repository, or if needed to look up the source revision number.

Switches

```
--message (-m) TEXT
--file (-F) FILE
--revision (-r) REV
--quiet (-q)
--username USER
--password PASS
```

```
--no-auth-cache
--non-interactive
--force-log
--editor-cmd EDITOR
--encoding ENC
--config-dir DIR
```

Examples

Copy an item within your working copy (just schedules the copy—nothing goes into the repository until you commit):

```
$ svn copy foo.txt bar.txt
A         bar.txt
$ svn status
A  +   bar.txt
```

Copy an item in your working copy to a URL in the repository (an immediate commit, so you must supply a commit message):

```
$ svn copy near.txt file:///tmp/repos/test/far-away.txt -m "Remote copy."

Committed revision 8.
```

Copy an item from the repository to your working copy (just schedules the copy—nothing goes into the repository until you commit):

Tip

This is the recommended way to resurrect a dead file in your repository!

```
$ svn copy file:///tmp/repos/test/far-away near-here
A         near-here
```

And finally, copying between two URLs:

```
$ svn copy file:///tmp/repos/test/far-away file:///tmp/repos/test/over-there -m "r

Committed revision 9.
```

Tip

This is the easiest way to “tag” a revision in your repository—just **svn copy** that revision (usually HEAD) into your tags directory.

```
$ svn copy file:///tmp/repos/test/trunk file:///tmp/repos/test/tags/0.6.32-prerele

Committed revision 12.
```

And don't worry if you forgot to tag—you can always specify an older revision and tag anytime:

```
$ svn copy -r 11 file:///tmp/repos/test/trunk file:///tmp/repos/test/tags/0.6.32-p
Committed revision 13.
```

Name

svn delete -- Delete an item from a working copy or the repository.

Synopsis

```
svn delete PATH...
```

```
svn delete URL...
```

Description

Items specified by *PATH* are scheduled for deletion upon the next commit. Files (and directories that have not been committed) are immediately removed from the working copy. The command will not remove any unversioned or modified items; use the `--force` switch to override this behavior.

Items specified by URL are deleted from the repository via an immediate commit. Multiple URLs are committed atomically.

Alternate Names

del, remove, rm

Changes

Working copy if operating on files, Repository if operating on URLs

Accesses Repository

Only if operating on URLs

Switches

```
--force
--force-log
--message (-m) TEXT
--file (-F) FILE
--quiet (-q)
--targets FILENAME
--username USER
--password PASS
--no-auth-cache
--non-interactive
--editor-cmd EDITOR
--encoding ENC
--config-dir DIR
```

Examples

Using **svn** to delete a file from your working copy merely schedules it to be deleted. When you commit, the file is deleted in the repository.

```
$ svn delete myfile
D         myfile

$ svn commit -m "Deleted file 'myfile'."
Deleting       myfile
Transmitting file data .
Committed revision 14.
```

Deleting a URL, however, is immediate, so you have to supply a log message:

```
$ svn delete -m "Deleting file 'yourfile'" file:///tmp/repos/test/yourfile

Committed revision 15.
```

Here's an example of how to force deletion of a file that has local mods:

```
$ svn delete over-there
svn: Attempting restricted operation for modified resource
svn: Use --force to override this restriction
svn: 'over-there' has local modifications

$ svn delete --force over-there
D         over-there
```

Name

svn diff -- Display the differences between two paths.

Synopsis

```
diff [-r N[:M]] [TARGET[@REV]...]
```

```
diff [-r N[:M]] --old OLD-TGT[@OLDREV] [--new NEW-TGT[@NEWREV]] [PATH...]
```

```
diff OLD-URL[@OLDREV] NEW-URL[@NEWREV]
```

Description

Display the differences between two paths. The three different ways you can use **svn diff** are:

svn diff [-r N[:M]] [--old OLD-TGT] [--new NEW-TGT] [PATH...] displays the differences between *`OLD-TGT`* and *`NEW-TGT`*. If *`PATH`*s are given, they are treated as relative to *`OLD-TGT`* and *`NEW-TGT`* and the output is restricted to differences in only those paths. *`OLD-TGT`* and *`NEW-TGT`* may be working copy paths or *`URL`*`[`*`@REV`*`]`. *`OLD-TGT`* defaults to the current working directory and *`NEW-TGT`* defaults to *`OLD-TGT`*. *`N`* defaults to `BASE` or, if *`OLD-TGT`* is a URL, to `HEAD`. *`M`* defaults to the current working version or, if *`NEW-TGT`* is a URL, to `HEAD`. **svn diff -r N** sets the revision of *`OLD-TGT`* to *`N`*, **svn diff -r N:M** also sets the revision of *`NEW-TGT`* to *`M`*.

svn diff -r N:M URL is shorthand for **svn diff -r N:M --old=URL --new=URL**.

svn diff [-r N[:M]] URL1[@N] URL2[@M] is shorthand for **svn diff [-r N[:M]] --old=URL1 --new=URL2**.

If *`TARGET`* is a URL, then revs N and M can be given either via the `--revision` or by using “@” notation as described earlier.

If *`TARGET`* is a working copy path, then the `--revision` switch means:

`--revision N:M`

The server compares /F10 10.0 Tf *1 0 0 1 130.083 340.399*

Alternate Names

di

Changes

Nothing

Accesses Repository

For obtaining differences against anything but BASE revision in your working copy

Switches

```
--revision (-r) REV
--old OLD-TARGET
--new NEW-TARGET
--extensions (-x) "ARGS"
--non-recursive (-N)
--diff-cmd CMD
--notice-ancestry
--username USER
--password PASS
--no-auth-cache
--non-interactive
--no-diff-deleted
--config-dir DIR
```

Examples

Compare BASE and your working copy (one of the most popular uses of **svn diff**):

```
$ svn diff COMMITTERS
Index: COMMITTERS
===================================================================
--- COMMITTERS (revision 4404)
+++ COMMITTERS (working copy)
```

See how your working copy's modifications compare against an older revision:

```
$ svn diff -r 3900 COMMITTERS
Index: COMMITTERS
===================================================================
--- COMMITTERS (revision 3900)
+++ COMMITTERS (working copy)
```

Compare revision 3000 to revision 3500 using “@” syntax:

```
$ svn diff http://svn.collab.net/repos/svn/trunk/COMMITTERS@3000 http://svn.collab
Index: COMMITTERS
===================================================================
--- COMMITTERS (revision 3000)
+++ COMMITTERS (revision 3500)
…
```

Compare revision 3000 to revision 3500 using range notation (you only pass the one URL in this case):

```
$ svn diff -r 3000:3500 http://svn.collab.net/repos/svn/trunk/COMMITTERS
Index: COMMITTERS
===================================================================
--- COMMITTERS (revision 3000)
+++ COMMITTERS (revision 3500)
```

Compare revision 3000 to revision 3500 of all files in `trunk` using range notation:

```
$ svn diff -r 3000:3500 http://svn.collab.net/repos/svn/trunk
```

Compare revision 3000 to revision 3500 of only three files in `trunk` using range notation:

```
$ svn diff -r 3000:3500 --old http://svn.collab.net/repos/svn/trunk COMMITTERS REA
```

If you have a working copy, you can obtain the differences without typing in the long URLs:

```
$ svn diff -r 3000:3500 COMMITTERS
Index: COMMITTERS
===================================================================
--- COMMITTERS (revision 3000)
+++ COMMITTERS (revision 3500)
```

Use `--diff-cmd` *`CMD`* `-x` to pass arguments directly to the external diff program

```
$ svn diff --diff-cmd /usr/bin/diff -x "-i -b" COMMITTERS
Index: COMMITTERS
===================================================================
0a1,2
> This is a test
>
```

Name

svn export -- Export a clean directory tree.

Synopsis

```
svn export [-r REV] URL[@PEGREV] [PATH]
```

```
svn export [-r REV] PATH1[@PEGREV] [PATH2]
```

Description

The first form exports a clean directory tree from the repository specified by URL, at revision *`REV`* if it is given, otherwise at `HEAD`, into *`PATH`*. If *`PATH`* is omitted, the last component of the *`URL`* is used for the local directory name.

The second form exports a clean directory tree from the working copy specified by *`PATH1`* into *`PATH2`*. All local changes will be preserved, but files not under version control will not be copied.

Alternate Names

None

Changes

Local disk

Accesses Repository

Only if exporting from a URL

Switches

```
--revision (-r) REV
--quiet (-q)
--force
--username USER
--password PASS
--no-auth-cache
--non-interactive
--non-recursive
--config-dir DIR
--native-eol EOL
--ignore-externals
```

Examples

Export from your working copy (doesn't print every file and directory):

```
$ svn export a-wc my-export
Export complete.
```

Export directly from the repository (prints every file and directory):

```
$ svn export file:///tmp/repos my-export
A  my-export/test
A  my-export/quiz
…
Exported revision 15.
```

When rolling operating-system-specific release packages, it can be useful to export a tree which uses a specific EOL character for line endings. The `--native-eol` option will do this, but it only affects files that have `svn:eol-style = native` properties attached to them. For example, to export a tree with all CRLF line endings (possibly for a Windows .zip file distribution):

```
$ svn export file://tmp/repos my-export --native-eol CRLF
A  my-export/test
A  my-export/quiz
…
Exported revision 15.
```

Name

svn help -- Help!

Synopsis

```
svn help [SUBCOMMAND...]
```

Description

This is your best friend when you're using Subversion and this book isn't within reach!

Alternate Names

?, h

Changes

Nothing

Accesses Repository

No

Switches

```
--version
--quiet (-q)
```

Name

svn import -- Commit an unversioned file or tree into the repository.

Synopsis

```
svn import [PATH] URL
```

Description

Recursively commit a copy of *PATH* to *URL*. If *PATH* is omitted “`.`” is assumed. Parent directories are created in the repository as necessary.

Alternate Names

None

Changes

Repository

Accesses Repository

Yes

Switches

```
--message (-m) TEXT
--file (-F) FILE
--quiet (-q)
--non-recursive (-N)
--username USER
--password PASS
--no-auth-cache
--non-interactive
--force-log
--editor-cmd EDITOR
--encoding ENC
--config-dir DIR
--auto-props
--no-auto-props
--ignore-externals
```

Examples

This imports the local directory `myproj` into the root of your repository:

```
$ svn import -m "New import" myproj http://svn.red-bean.com/repos/test
Adding         myproj/sample.txt
…
Transmitting file data .........
Committed revision 16.
```

This imports the local directory `myproj` into `trunk/misc` in your repository. The directory `trunk/misc` need not exist before you import into it—**svn import** will recursively create directories for you:

```
$ svn import -m "New import" myproj \
    http://svn.red-bean.com/repos/test/trunk/misc/myproj
Adding         myproj/sample.txt
…
Transmitting file data .........
Committed revision 19.
```

After importing data, note that the original tree is *not* under version control. To start working, you still need to **svn checkout** a fresh working copy of the tree.

Name

svn info -- Display information about a local or remote item.

Synopsis

```
svn info [TARGET...]
```

Description

Print information about both working copy paths and URLs, including:

- Path
- Name
- URL
- Revision
- Repository Root
- Repository UUID
- Node Kind
- Last Changed Author
- Last Changed Revision
- Last Changed Date
- Text Last Updated
- Properties Last Updated
- Checksum
- Lock Token
- Lock Owner
- Lock Creation Date

Alternate Names

None

Changes

Nothing

Accesses Repository

Only if operating on URLs

Switches

```
--targets FILENAME
--recursive (-R)
--revision (-r)
--config-dir DIR
```

Examples

svn info will show you all the useful information that it has for items in your working copy. It will show information for files:

```
$ svn info foo.c
Path: foo.c
Name: foo.c
URL: http://svn.red-bean.com/repos/test/foo.c
Repository Root: http://svn.red-bean.com/repos/test
Repository UUID: 5e7d134a-54fb-0310-bd04-b611643e5c25
Revision: 4417
Node Kind: file
Schedule: normal
Last Changed Author: sally
Last Changed Rev: 20
Last Changed Date: 2003-01-13 16:43:13 -0600 (Mon, 13 Jan 2003)
Text Last Updated: 2003-01-16 21:18:16 -0600 (Thu, 16 Jan 2003)
Properties Last Updated: 2003-01-13 21:50:19 -0600 (Mon, 13 Jan 2003)
Checksum: /3L38YwzhT93BWvgpdF6Zw==
```

It will also show information for directories:

```
$ svn info vendors
Path: vendors
URL: http://svn.red-bean.com/repos/test/vendors
Repository Root: http://svn.red-bean.com/repos/test
Repository UUID: 5e7d134a-54fb-0310-bd04-b611643e5c25
Revision: 19
Node Kind: directory
Schedule: normal
Last Changed Author: harry
Last Changed Rev: 19
Last Changed Date: 2003-01-16 23:21:19 -0600 (Thu, 16 Jan 2003)
```

svn info also acts on URLs (also note that the file readme.doc in this example is locked, so lock information is also provided):

```
$ svn info http://svn.red-bean.com/repos/test/readme.doc
Path: readme.doc
Name: readme.doc
URL: http://svn.red-bean.com/repos/test/readme.doc
Repository Root: http://svn.red-bean.com/repos/test
Repository UUID: 5e7d134a-54fb-0310-bd04-b611643e5c25
Revision: 1
Node Kind: file
```

```
Schedule: normal
Last Changed Author: sally
Last Changed Rev: 42
Last Changed Date: 2003-01-14 23:21:19 -0600 (Tue, 14 Jan 2003)
Text Last Updated: 2003-01-14 23:21:19 -0600 (Tue, 14 Jan 2003)
Checksum: d41d8cd98f00b204e9800998ecf8427e
Lock Token: opaquelocktoken:14011d4b-54fb-0310-8541-dbd16bd471b2
Lock Owner: harry
Lock Created: 2003-01-15 17:35:12 -0600 (Wed, 15 Jan 2003)
```

Name

svn list -- List directory entries in the repository.

Synopsis

```
svn list [TARGET[@REV]...]
```

Description

List each *TARGET* file and the contents of each *TARGET* directory as they exist in the repository. If *TARGET* is a working copy path, the corresponding repository URL will be used.

The default *TARGET* is “`.`”, meaning the repository URL of the current working copy directory.

With `--verbose`, the following fields show the status of the item:

- Revision number of the last commit
- Author of the last commit
- Size (in bytes)
- Date and time of the last commit

With `--xml`, output is in XML format (with a header and an enclosing document element unless `--incremental` is also specified). All of the information is present; the `--verbose` option is not accepted.

Alternate Names

ls

Changes

Nothing

Accesses Repository

Yes

Switches

```
--revision (-r) REV
--verbose (-v)
--recursive (-R)
--incremental
--xml
--username USER
--password PASS
--no-auth-cache
--non-interactive
--config-dir DIR
```

Examples

svn list is most useful if you want to see what files a repository has without downloading a working copy:

```
$ svn list http://svn.red-bean.com/repos/test/support
README.txt
INSTALL
examples/
…
```

You can pass the `--verbose` switch for additional information, rather like the UNIX command **ls -l**:

```
$ svn list --verbose file:///tmp/repos
     16 sally         28361 Jan 16 23:18 README.txt
     27 sally             0 Jan 18 15:27 INSTALL
     24 harry               Jan 18 11:27 examples/
```

For further details, see the section called “**svn list**”.

Name

svn lock -- Lock working copy paths or URLs in the repository, so that no other user can commit changes to them.

Synopsis

```
svn lock TARGET...
```

Description

Lock each *`TARGET`*. If any *`TARGET`* is already locked by another user, print a warning and continue locking the rest of the *`TARGET`*s. Use `--force` to steal a lock from another user or working copy.

Alternate Names

None

Changes

Working Copy, Repository

Accesses Repository

Yes

Switches

```
--targets ARG
--message (-m) ARG
--file (-F) ARG
--force-log
--encoding ARG
--username ARG
--password ARG
--no-auth-cache
--non-interactive
--config-dir ARG
--force
```

Examples

Lock two files in your working copy:

```
$ svn lock tree.jpg house.jpg
'tree.jpg' locked by user 'harry'.
'house.jpg' locked by user 'harry'.
```

Lock a file in your working copy that is currently locked by another user:

```
$ svn lock tree.jpg
```

```
svn: warning: Path '/tree.jpg is already locked by user 'harry in \
     filesystem '/svn/repos/db'

$ svn lock --force foo
'tree.jpg' locked by user 'sally'.
```

Lock a file without a working copy:

```
$ svn lock http://svn.red-bean.com/repos/test/tree.jpg
'tree.jpg' locked by user 'sally'.
```

Name

svn log -- Display commit log messages.

Synopsis

```
svn log [PATH]
```

```
svn log URL [PATH...]
```

Description

The default target is the path of your current directory. If no arguments are supplied, **svn log** shows the log messages for all files and directories inside of (and including) the current working directory of your working copy. You can refine the results by specifying a path, one or more revisions, or any combination of the two. The default revision range for a local path is `BASE:1`.

If you specify a URL alone, then it prints log messages for everything that the URL contains. If you add paths past the URL, only messages for those paths under that URL will be printed. The default revision range for a URL is `HEAD:1`.

With `--verbose`, **svn log** will also print all affected paths with each log message. With `--quiet`, **svn log** will not print the log message body itself (this is compatible with `--verbose`).

Each log message is printed just once, even if more than one of the affected paths for that revision were explicitly requested. Logs follow copy history by default. Use `--stop-on-copy` to disable this behavior, which can be useful for determining branch points.

Alternate Names

None

Changes

Nothing

Accesses Repository

Yes

Switches

```
--revision (-r) REV
--quiet (-q)
--verbose (-v)
--targets FILENAME
--stop-on-copy
--incremental
--limit NUM
--xml
--username USER
--password PASS
--no-auth-cache
--non-interactive
```

```
--config-dir DIR
```

Examples

You can see the log messages for all the paths that changed in your working copy by running **svn log** from the top:

```
$ svn log
------------------------------------------------------------------------
r20 | harry | 2003-01-17 22:56:19 -0600 (Fri, 17 Jan 2003) | 1 line

Tweak.
------------------------------------------------------------------------
r17 | sally | 2003-01-16 23:21:19 -0600 (Thu, 16 Jan 2003) | 2 lines
…
```

Examine all log messages for a particular file in your working copy:

```
$ svn log foo.c
------------------------------------------------------------------------
r32 | sally | 2003-01-13 00:43:13 -0600 (Mon, 13 Jan 2003) | 1 line

Added defines.
------------------------------------------------------------------------
r28 | sally | 2003-01-07 21:48:33 -0600 (Tue, 07 Jan 2003) | 3 lines
…
```

If you don't have a working copy handy, you can log a URL:

```
$ svn log http://svn.red-bean.com/repos/test/foo.c
------------------------------------------------------------------------
r32 | sally | 2003-01-13 00:43:13 -0600 (Mon, 13 Jan 2003) | 1 line

Added defines.
------------------------------------------------------------------------
r28 | sally | 2003-01-07 21:48:33 -0600 (Tue, 07 Jan 2003) | 3 lines
…
```

`-incremental` switch. **svn log** normally prints out a dashed line at the beginning of a log message, after each subsequent log message, and following the final log message. If you ran **svn log** on a range of two revisions, you would get this:

```
$ svn log -r 14:15
------------------------------------------------------------------------
r14 | ...

------------------------------------------------------------------------
r15 | ...

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

However, if you wanted to gather 2 non-sequential log messages into a file, you might do something like this:

```
$ svn log -r 14 > mylog
$ svn log -r 19 >> mylog
$ svn log -r 27 >> mylog
$ cat mylog
------------------------------------------------------------------------
r14 | ...

------------------------------------------------------------------------
------------------------------------------------------------------------
r19 | ...

------------------------------------------------------------------------
------------------------------------------------------------------------
r27 | ...

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

You can avoid the clutter of the double dashed lines in your output by using the incremental switch:

```
$ svn log --incremental -r 14 > mylog
$ svn log --incremental -r 19 >> mylog
$ svn log --incremental -r 27 >> mylog
$ cat mylog
------------------------------------------------------------------------
r14 | ...

------------------------------------------------------------------------
r19 | ...

------------------------------------------------------------------------
r27 | ...
```

The `--incremental` switch provides similar output control when using the `--xml` switch.

Tip

If you run **svn log** on a specific path and provide a specific revision and get no output at all

```
$ svn log -r 20 http://svn.red-bean.com/untouched.txt
------------------------------------------------------------------------
```

That just means that the path was not modified in that revision. If you log from the top of the repository, or know the file that changed in that revision, you can specify it explicitly:

```
$ svn log -r 20 touched.txt
------------------------------------------------------------------------
r20 | sally | 2003-01-17 22:56:19 -0600 (Fri, 17 Jan 2003) | 1 line

Made a change.
------------------------------------------------------------------------
```

Name

svn merge -- Apply the differences between two sources to a working copy path.

Synopsis

```
svn merge sourceURL1[@N] sourceURL2[@M] [WCPATH]
```

```
svn merge sourceWCPATH1@N sourceWCPATH2@M [WCPATH]
```

```
svn merge -r N:M SOURCE[@REV] [WCPATH]
```

Description

In the first and second forms, the source paths (URLs in the first form, working copy paths in the second) are specified at revisions *N* and *M*. These are the two sources to be compared. The revisions default to HEAD if omitted.

In the third form, *SOURCE* can be a URL or working copy item, in which case the corresponding URL is used. This URL, at revisions *N* and

```
--non-interactive
--config-dir DIR
```

Examples

Merge a branch back into the trunk (assuming that you have a working copy of the trunk, and that the branch was created in revision 250):

```
$ svn merge -r 250:HEAD http://svn.red-bean.com/repos/branches/my-branch
U  myproj/tiny.txt
U  myproj/thhgttg.txt
U  myproj/win.txt
U  myproj/flo.txt
```

If you branched at revision 23, and you want to merge changes on trunk into your branch, you could do this from inside the working copy of your branch:

```
$ svn merge -r 23:30 file:///tmp/repos/trunk/vendors
U  myproj/thhgttg.txt
…
```

To merge changes to a single file:

```
$ cd myproj
$ svn merge -r 30:31 thhgttg.txt
U  thhgttg.txt
```

Name

svn mkdir -- Create a new directory under version control.

Synopsis

```
svn mkdir PATH...
```

```
svn mkdir URL...
```

Description

Create a directory with a name given by the final component of the *PATH* or URL. A directory specified by a working copy *PATH* is scheduled for addition in the working copy. A directory specified by a URL is created in the repository via an immediate commit. Multiple directory URLs are committed atomically. In both cases all the intermediate directories must already exist.

Alternate Names

None

Changes

Working copy, repository if operating on a URL

Accesses Repository

Only if operating on a URL

Switches

```
--message (-m) TEXT
--file (-F) FILE
--quiet (-q)
--username USER
--password PASS
--no-auth-cache
--non-interactive
--editor-cmd EDITOR
--encoding ENC
--force-log
--config-dir DIR
```

Examples

Create a directory in your working copy:

```
$ svn mkdir newdir
A         newdir
```

Create one in the repository (instant commit, so a log message is required):

```
$ svn mkdir -m "Making a new dir." http://svn.red-bean.com/repos/newdir
Committed revision 26.
```

Name

svn move -- Move a file or directory.

Synopsis

```
svn move SRC DST
```

Description

This command moves a file or directory in your working copy or in the repository.

Tip

This command is equivalent to an **svn copy** followed by **svn delete**.

Note

Subversion does not support moving between working copies and URLs. In addition, you can only move files within a single repository—Subversion does not support cross-repository moving.

WC -> WC
Move and schedule a file or directory for addition (with history).

URL -> URL
Complete server-side rename.

Alternate Names

mv, rename, ren

Changes

Working copy, repository if operating on a URL

Accesses Repository

Only if operating on a URL

Switches

```
--message (-m) TEXT
--file (-F) FILE
--revision (-r) REV
--quiet (-q)
--force
--username USER
--password PASS
--no-auth-cache
```

```
--non-interactive
--editor-cmd EDITOR
--encoding ENC
--force-log
--config-dir DIR
```

Examples

Move a file in your working copy:

```
$ svn move foo.c bar.c
A         bar.c
D         foo.c
```

Move a file in the repository (an immediate commit, so it requires a commit message):

```
$ svn move -m "Move a file" http://svn.red-bean.com/repos/foo.c \
                            http://svn.red-bean.com/repos/bar.c

Committed revision 27.
```

Name

svn propdel -- Remove a property from an item.

Synopsis

```
svn propdel PROPNAME [PATH...]
```

```
svn propdel PROPNAME --revprop -r REV [URL]
```

Description

This removes properties from files, directories, or revisions. The first form removes versioned properties in your working copy, while the second removes unversioned remote properties on a repository revision.

Alternate Names

pdel, pd

Changes

Name

svn propedit -- Edit the property of one or more items under version control.

Synopsis

```
svn propedit PROPNAME PATH...
```

```
svn propedit PROPNAME --revprop -r REV [URL]
```

Description

Edit one or more properties using your favorite editor. The first form edits versioned properties in your working copy, while the second edits unversioned remote properties on a repository revision.

Alternate Names

pedit, pe

Changes

Working copy, repository only if operating on a URL

Accesses Repository

Only if operating on a URL

Switches

```
--revision (-r) REV
--revprop
--username USER
--password PASS
--no-auth-cache
--non-interactive
--encoding ENC
--editor-cmd EDITOR
--config-dir DIR
```

Examples

svn propedit makes it easy to modify properties that have multiple values:

```
$ svn propedit svn:keywords  foo.c
    <svn will launch your favorite editor here, with a buffer open
    containing the current contents of the svn:keywords property.  You
    can add multiple values to a property easily here by entering one
    value per line.>
Set new value for property 'svn:keywords' on 'foo.c'
```

Name

svn propget -- Print the value of a property.

Synopsis

```
svn propget PROPNAME [TARGET[@REV]...]
```

```
svn propget PROPNAME --revprop -r REV [URL]
```

Description

Print the value of a property on files, directories, or revisions. The first form prints the versioned property of an item or items in your working copy, while the second prints unversioned remote property on a repository revision. See the section called “Properties” for more information on properties.

Alternate Names

pget, pg

Changes

Working copy, repository only if operating on a URL

Accesses Repository

Only if operating on a URL

Switches

```
--recursive (-R)
--revision (-r) REV
--revprop
--strict
--username USER
--password PASS
--no-auth-cache
--non-interactive
--config-dir DIR
```

Examples

Examine a property of a file in your working copy:

```
$ svn propget svn:keywords foo.c
Author
```

```
$ svn propget svn:log --revprop -r 20
Began journal.
```

Name

svn proplist -- List all properties.

Synopsis

```
svn proplist [TARGET[@REV]...]
```

```
svn proplist --revprop -r REV [URL]
```

Description

List all properties on files, directories, or revisions. The first form lists versioned properties in your working copy, while the second lists unversioned remote properties on a repository revision.

Alternate Names

plist, pl

Changes

Working copy, repository only if operating on a URL

Accesses Repository

Only if operating on a URL

Switches

```
--verbose (-v)
--recursive (-R)
--revision (-r) REV
--quiet (-q)
--revprop
--username USER
--password PASS
--no-auth-cache
--non-interactive
--config-dir DIR
```

Examples

You can use proplist to see the properties on an item in your working copy:

```
$ svn proplist foo.c
Properties on 'foo.c':
  svn:mime-type
  svn:keywords
  owner
```

But with the `--verbose` flag, svn proplist is extremely handy as it also shows you the values for the properties:

```
$ svn proplist --verbose foo.c
Properties on 'foo.c':
  svn:mime-type : text/plain
  svn:keywords : Author Date Rev
  owner : sally
```

Name

svn propset -- Set PROPNAME to PROPVAL on files, directories, or revisions.

Synopsis

```
svn propset PROPNAME [PROPVAL | -F VALFILE] PATH...
```

```
svn propset PROPNAME --revprop -r REV [PROPVAL | -F VALFILE] [URL]
```

Description

Set *`PROPNAME`* to *`PROPVAL`* on files, directories, or revisions. The first example creates a versioned, local property change in the working copy, and the second creates an unversioned, remote property change on a repository revision.

Tip

Subversion has a number of “special” properties that affect its behavior. See the section called “Special Properties” for more on these properties.

Alternate Names

pset, ps

Changes

Working copy, repository only if operating on a URL

Accesses Repository

Only if operating on a URL

Switches

```
--file (-F) FILE
--quiet (-q)
--revision (-r) REV
--targets FILENAME
--recursive (-R)
--revprop
--username USER
--password PASS
--no-auth-cache
--non-interactive
--encoding ENC
--force
--config-dir DIR
```

Examples

Set the mimetype on a file:

```
$ svn propset svn:mime-type image/jpeg foo.jpg
property 'svn:mime-type' set on 'foo.jpg'
```

On a UNIX system, if you want a file to have the executable permission set:

```
$ svn propset svn:executable ON somescript
property 'svn:executable' set on 'somescript'
```

Perhaps you have an internal policy to set certain properties for the benefit of your coworkers:

```
$ svn propset owner sally foo.c
property 'owner' set on 'foo.c'
```

If you made a mistake in a log message for a particular revision and want to change it, use `--revprop` and set `svn:log` to the new log message:

```
$ svn propset --revprop -r 25 svn:log "Journaled about trip to New York."
property 'svn:log' set on repository revision '25'
```

Name

svn resolved -- Remove “conflicted” state on working copy files or directories.

Synopsis

```
svn resolved PATH...
```

Description

Remove “conflicted” state on working copy files or directories. This routine does not semantically resolve conflict markers; it merely removes conflict-related artifact files and allows *PATH* to be committed again; that is, it tells Subversion that the conflicts have been “resolved”. See the section called “Resolve Conflicts (Merging Others' Changes)” for an in-depth look at resolving conflicts.

Alternate Names

None

Changes

Working copy

Accesses Repository

No

Switches

```
--targets FILENAME
--recursive (-R)
--quiet (-q)
--config-dir DIR
```

Examples

If you get a conflict on an update, your working copy will sprout three new files:

```
$ svn update
C  foo.c
Updated to revision 31.
$ ls
foo.c
foo.c.mine
foo.c.r30
foo.c.r31
```

Once you've resolved the conflict and `foo.c` is ready to be committed, run **svn resolved** to let your working copy know you've taken care of everything.

Warning

You *can* just remove the conflict files and commit, but **svn resolved** fixes up some bookkeeping data in the working copy administrative area in addition to removing the conflict files, so we recommend that you use this command.

Name

svn revert -- Undo all local edits.

Synopsis

```
svn revert PATH...
```

Description

Reverts any local changes to a file or directory and resolves any conflicted states. **svn revert** will not only revert the contents of an item in your working copy, but also any property changes. Finally, you can use it to undo any scheduling operations that you may have done (e.g. files scheduled for addition or deletion can be "unscheduled").

Alternate Names

None

Changes

Working copy

Accesses Repository

No

Switches

```
--targets FILENAME
--recursive (-R)
--quiet (-q)
--config-dir DIR
```

Examples

Discard changes to a file:

```
$ svn revert foo.c
Reverted foo.c
```

If you want to revert a whole directory of files, use the `--recursive` flag:

```
$ svn revert --recursive .
Reverted newdir/afile
Reverted foo.c
Reverted bar.txt
```

Lastly, you can undo any scheduling operations:

```
$ svn add mistake.txt whoops
A         mistake.txt
A         whoops
A         whoops/oopsie.c

$ svn revert mistake.txt whoops
Reverted mistake.txt
Reverted whoops

$ svn status
?      mistake.txt
?      whoops
```

Note

If you provide no targets to **svn revert**, it will do nothing—to protect you from accidentally losing changes in your working copy, **svn revert** requires you to provide at least one target.

Name

svn status -- Print the status of working copy files and directories.

Synopsis

```
svn status [PATH...]
```

Description

Print the status of working copy files and directories. With no arguments, it prints only locally modified items (no repository access). With `--show-updates`, add working revision and server out-of-date information. With `--verbose`, print full revision information on every item.

The first six columns in the output are each one character wide, and each column gives you information about different aspects of each working copy item.

The first column indicates that an item was added, deleted, or otherwise changed.

' '
No modifications.

'A'
Item is scheduled for Addition.

'D'
Item is scheduled for Deletion.

'M'
Item has been modified.

'R'
Item has been replaced in your working copy. This means the file was scheduled for deletion, and then a new file with the same name was scheduled for addition in its place.

'C'
The contents (as opposed to the properties) of the item conflict with updates received from the repository.

'X'
Item is related to an externals definition.

'I'
Item is being ignored (e.g. with the `svn:ignore` property).

'?'
Item is not under version control.

'!'
Item is missing (e.g. you moved or deleted it without using **svn**). This also indicates that a directory is incomplete (a checkout or update was interrupted).

'~'
Item is versioned as one kind of object (file, directory, link), but has been replaced by different kind of object.

The second column tells the status of a file's or directory's properties.

' '
No modifications.

'M'
Properties for this item have been modified.

'C'
Properties for this item are in conflict with property updates received from the repository.

The third column is populated only if the working copy directory is locked. (See the section called "**svn cleanup**".)

' '
Item is not locked.

'L'
Item is locked.

The fourth column is populated only if the item is scheduled for addition-with-history.

' '
No history scheduled with commit.

'+'
History scheduled with commit.

The fifth column is populated only if the item is switched relative to its parent (see the section called "Switching a Working Copy").

' '
Item is a child of its parent directory.

'S'
Item is switched.

The sixth column is populated with lock information.

' '
When `--show-updates` is used, the file is not locked. If `--show-updates` is *not* used, this merely means that the file is not locked in this working copy.

K
File is locked in this working copy.

O
File is locked either by another user or in another working copy. This only appears when `--show-updates` is used.

T

File was locked in this working copy, but the lock has been “stolen”and is invalid. The file is currently locked in the repository. This only appears when `--show-updates` is used.

B

```
 M     wc/bar.c
A  +   wc/qax.c
```

If you want to find out what files in your working copy are out-of-date, pass the `--show-updates` switch (this will *not* make any changes to your working copy). Here you can see that `wc/foo.c` has changed in the repository since we last updated our working copy:

```
$ svn status --show-updates wc
 M           965    wc/bar.c
       *     965    wc/foo.c
A  +         965    wc/qax.c
Status against revision:    981
```

Name

And to switch back, just provide the URL to the location in the repository from which you originally checked out your working copy:

```
$ svn switch http://svn.red-bean.com/repos/trunk/vendors .
U  myproj/foo.txt
U  myproj/bar.txt
U  myproj/baz.c
U  myproj/qux.c
Updated to revision 31.
```

Tip

You can just switch part of your working copy to a branch if you don't want to switch your entire working copy.

Sometimes an administrator might change the “base location” of your repository—in other words, the contents of the repository doesn't change, but the main URL used to reach the root of the repository does. For example, the hostname may change, the URL scheme, or any part of the URL which leads to the repository itself. Rather than checkout a new working copy, you can have the **svn switch** command “rewrite” the beginnings of all the URLs in your working copy. Use the `--relocate` option to do the substitution. No file contents are changed, nor is the repository contacted. It's similar to running a Perl script over your working copy `.svn/` directories which runs **s/OldRoot/NewRoot/**.

Name

svn unlock -- Unlock working copy paths or URLs.

Synopsis

```
svn unlock TARGET...
```

Description

Unlock each *TARGET*. If any *TARGET* is either locked by another user or no valid lock token exists in the working copy, print a warning and continue unlocking the rest of the *TARGET*s. Use `--force` to break a lock belonging to another user or working copy.

Alternate Names

None

Changes

Working Copy, Repository

Accesses Repository

Yes

Switches

```
--targets ARG
--username ARG
--password ARG
--no-auth-cache
--non-interactive
--config-dir ARG
--force
```

Examples

Unlock two files in your working copy:

```
$ svn unlock tree.jpg house.jpg
'tree.jpg' unlocked.
'house.jpg' unlocked.
```

Unlock a file in your working copy that is currently locked by another user:

```
$ svn unlock tree.jpg
svn: 'tree.jpg' is not locked in this working copy
$ svn unlock --force tree.jpg
'tree.jpg' unlocked.
```

Unlock a file without a working copy:

```
$ svn unlock http://svn.red-bean.com/repos/test/tree.jpg
'tree.jpg unlocked.
```

For further details, see the section called “Locking”.

Name

svn update -- Update your working copy.

Synopsis

```
svn update [PATH...]
```

Description

svn update brings changes from the repository into your working copy. If no revision given, it brings your working copy up-to-date with the `HEAD` revision. Otherwise, it synchronizes the working copy to the revision given by the `--revision` switch. As part of the synchronization, **svn update** also removes any stale locks (see the section called “**svn cleanup**”) found in the working copy.

For each updated item a line will start with a character reporting the action taken. These characters have the following meaning:

A
Added

D
Deleted

U
Updated

C
Conflict

G
Merged

A character in the first column signifies an update to the actual file, while updates to the file's properties are shown in the second column.

Alternate Names

up

Changes

Working copy

Accesses Repository

Yes

Switches

```
--revision (-r) REV
--non-recursive (-N)
--quiet (-q)
```

```
--diff3-cmd CMD
--username USER
--password PASS
--no-auth-cache
--non-interactive
--config-dir DIR
--ignore-externals
```

Examples

Pick up repository changes that have happened since your last update:

```
$ svn update
A  newdir/toggle.c
A  newdir/disclose.c
A  newdir/launch.c
D  newdir/README
Updated to revision 32.
```

You can also update your working copy to an older revision (Subversion doesn't have the concept of “sticky” files like CVS does; see Appendix A, *Subversion for CVS Users*):

```
$ svn update -r30
A  newdir/README
D  newdir/toggle.c
D  newdir/disclose.c
D  newdir/launch.c
U  foo.c
Updated to revision 30.
```

> **Tip**
>
> If you want to examine an older revision of a single file, you may want to use **svn cat**.

svnadmin

svnadmin is the administrative tool for monitoring and repairing your Subversion repository. For detailed information, see the section called “svnadmin”.

Since **svnadmin** works via direct repository access (and thus can only be used on the machine that holds the repository), it refers to the repository with a path, not a URL.

svnadmin Switches

`--bdb-log-keep`
(Berkeley DB specific) Disable automatic log removal of database log files.

`--bdb-txn-nosync`
(Berkeley DB specific) Disables fsync when committing database transactions.

`--bypass-hooks`
Bypass the repository hook system.

`--clean-logs`
Removes unused Berkeley DB logs.

`--force-uuid`
By default, when loading data into repository that already contains revisions, **svnadmin** will ignore the `UUID` from the dump stream. This switch will cause the repository's `UUID` to be set to the `UUID` from the stream.

`--ignore-uuid`
By default, when loading an empty repository, **svnadmin** will use the `UUID` from the dump stream. This switch will cause that UUID to be ignored.

`--incremental`
Dump a revision only as a diff against the previous revision, instead of the usual fulltext.

`--parent-dir` *`DIR`*
When loading a dump file, root paths at *`DIR`* instead of `/`.

`--revision` (`-r`) *`ARG`*
Specify a particular revision to operate on.

`--quiet`
Do not show normal progress—show only errors.

`--use-post-commit-hook`
When loading a dump file, run the repository's post-commit hook after finalizing each newly loaded revision.

`--use-pre-commit-hook`
When loading a dump file, run the repository's pre-commit hook before finalizing each newly loaded revision. If the hook fails, abort the commit and terminate the load process.

svnadmin Subcommands

Name

svnadmin create -- Create a new, empty repository.

Synopsis

```
svnadmin create REPOS_PATH
```

Description

Create a new, empty repository at the path provided. If the provided directory does not exist, it will be created for you.[1] As of Subversion 1.2, **svnadmin** creates new repositories with the `fsfs` filesystem backend by default.

Switches

```
--bdb-txn-nosync
--bdb-log-keep
--config-dir DIR
```

[1]Remember, **svnadmin** works only with local *paths*, not *URLs*.

Name

svnadmin deltify -- Deltify changed paths in a revision range.

Synopsis

```
svnadmin deltify [-r LOWER[:UPPER]] REPOS_PATH
```

Description

svnadmin deltify only exists in 1.0.x due to historical reasons. This command is deprecated and no longer needed.

It dates from a time when Subversion offered administrators greater control over compression strategies in the repository. This turned out to be a lot of complexity for *very* little gain, and this “feature” was deprecated.

Switches

```
--revision (-r)
--quiet
```

Name

svnadmin dump -- Dump the contents of filesystem to stdout.

Synopsis

```
svnadmin dump REPOS_PATH [-r LOWER[:UPPER]] [--incremental]
```

Description

Dump the contents of filesystem to stdout in a “dumpfile” portable format, sending feedback to stderr. Dump revisions *`LOWER`* rev through *`UPPER`* rev. If no revisions are given, dump all revision trees. If only *`LOWER`* is given, dump that one revision tree. See the section called “Migrating a Repository” for a practical use.

By default, the Subversion dumpfile stream contains a single revision (the first revision in the requested

…

Incrementally dump a single transaction from your repository:

```
$ svnadmin dump /usr/local/svn/repos -r 21 --incremental
* Dumped revision 21.
SVN-fs-dump-format-version: 1
Revision-number: 21
Prop-content-length: 101
Content-length: 101
…
```

Name

svnadmin help

Synopsis

```
svnadmin help [SUBCOMMAND...]
```

Description

This subcommand is useful when you're trapped on a desert island with neither a net connection nor a copy of this book.

Alternate Names

?, h

Name

svnadmin hotcopy -- Make a hot copy of a repository.

Synopsis

```
svnadmin hotcopy REPOS_PATH NEW_REPOS_PATH
```

Description

This subcommand makes a full “hot” backup of your repository, including all hooks, configuration files, and, of course, database files. If you pass the `--clean-logs` switch, **svnadmin** will perform a hotcopy of your repository, and then remove unused Berkeley DB logs from the original repository. You can run this command at any time and make a safe copy of the repository, regardless of whether other processes are using the repository.

Switches

```
--clean-logs
```

Name

svnadmin list-dblogs -- Ask Berkeley DB which log files exist for a given Subversion repository (applies only to repositories using the `bdb` backend).

Synopsis

```
svnadmin list-dblogs REPOS_PATH
```

Description

Berkeley DB creates logs of all changes to the repository, which allow it to recover in the face of catastrophe. Unless you enable `DB_LOG_AUTOREMOVE`, the log files accumulate, although most are no longer used and can be deleted to reclaim disk space. See the section called “Managing Disk Space” for more information.

Name

svnadmin list-unused-dblogs -- Ask Berkeley DB which log files can be safely deleted (applies only to repositories using the `bdb` backend).

Synopsis

```
svnadmin list-unused-dblogs REPOS_PATH
```

Description

Berkeley DB creates logs of all changes to the repository, which allow it to recover in the face of catastrophe. Unless you enable `DB_LOG_AUTOREMOVE`, the log files accumulate, although most are no longer used and can be deleted to reclaim disk space. See the section called "Managing Disk Space" for more information.

Examples

Remove all unused log files from a repository:

```
$ svnadmin list-unused-dblogs /path/to/repos
/path/to/repos/log.0000000031
/path/to/repos/log.0000000032
/path/to/repos/log.0000000033

$ svnadmin list-unused-dblogs /path/to/repos | xargs rm
## disk space reclaimed!
```

Name

svnadmin load -- Read a “dumpfile”-formatted stream from stdin.

Synopsis

```
svnadmin load REPOS_PATH
```

Description

Read a “dumpfile”-formatted stream from stdin, committing new revisions into the repository's filesystem. Send progress feedback to stdout.

Switches

```
--quiet (-q)
--ignore-uuid
--force-uuid
--use-pre-commit-hook
--use-post-commit-hook
--parent-dir
```

Example

This shows the beginning of loading a repository from a backup file (made, of course, with **svnadmin dump**):

```
$ svnadmin load /usr/local/svn/restored < repos-backup
<<< Started new txn, based on original revision 1
     * adding path : test ... done.
     * adding path : test/a ... done.
…
```

Or if you want to load into a subdirectory:

```
$ svnadmin load --parent-dir new/subdir/for/project /usr/local/svn/restored < repo
<<< Started new txn, based on original revision 1
     * adding path : test ... done.
     * adding path : test/a ... done.
…
```

Name

svnadmin lslocks -- Print descriptions of all locks.

Synopsis

```
svnadmin lslocks REPOS_PATH
```

Description

Print descriptions of all locks in a repository.

Switches

None

Example

This lists the one locked file in the repository at `/svn/repos`

```
$ svnadmin lslocks /svn/repos
Path: /tree.jpg
UUID Token: opaquelocktoken:ab00ddf0-6afb-0310-9cd0-dda813329753
Owner: harry
Created: 2005-07-08 17:27:36 -0500 (Fri, 08 Jul 2005)
Expires:
Comment (1 line):
Rework the uppermost branches on the bald cypress in the foreground.
```

Name

svnadmin lstxns -- Print the names of all uncommitted transactions.

Synopsis

```
svnadmin lstxns REPOS_PATH
```

Description

Print the names of all uncommitted transactions. See the section called “Repository Cleanup” for information on how uncommitted transactions are created and what you should do with them.

Examples

List all outstanding transactions in a repository.

```
$ svnadmin lstxns /usr/local/svn/repos/
1w
1x] TJ 1 whu829998 (cdtransactions)  1 120.0 455.571 Tm -383.199298 (the) - .6Ooy5
```

Name

svnadmin recover -- Bring a repository database back into a consistent state (applies only to repositories using the `bdb` backend). In addition, if `repos/conf/passwd` does not exist, it will create a default password file .

Synopsis

```
svnadmin recover REPOS_PATH
```

Description

Run this command if you get an error indicating that your repository needs to be recovered.

Switches

```
--wait
```

Examples

Recover a hung repository:

```
$ svnadmin recover /usr/local/svn/repos/
Repository lock acquired.
Please wait; recovering the repository may take some time...

Recovery completed.
The latest repos revision is 34.
```

Recovering the database requires an exclusive lock on the repository. (This is a “database lock”; see Three meanings of “lock”.) If another process is accessing the repository, then **svnadmin recover** will error:

```
$ svnadmin recover /usr/local/svn/repos
svn: Failed to get exclusive repository access; perhaps another process
such as httpd, svnserve or svn has it open?

$
```

The `--wait` option, however, will cause **svnadmin recover** to wait indefinitely for other processes to disconnect:

```
$ svnadmin recover /usr/local/svn/repos --wait
Waiting on repository lock; perhaps another process has it open?

### time goes by...

Repository lock acquired.
Please wait; recovering the repository may take some time...

Recovery completed.
```

```
The latest repos revision is 34.
```

Name

svnadmin rmlocks -- Unconditionally remove one or more locks from a repository.

Synopsis

```
svnadmin rmlocks REPOS_PATH LOCKED_PATH...
```

Description

Remove lock from each *LOCKED_PATH*.

Switches

None

Example

This deletes the locks on `tree.jpg` and `house.jpg` in the repository at `/svn/repos`

```
$ svnadmin rmlocks /svn/repos tree.jpg house.jpg
Removed lock on '/tree.jpg.
Removed lock on '/house.jpg.
```

Name

svnadmin rmtxns -- Delete transactions from a repository.

Synopsis

```
svnadmin rmtxns REPOS_PATH TXN_NAME...
```

Description

Delete outstanding transactions from a repository. This is covered in detail in the section called “Repository Cleanup”.

Switches

```
--quiet (-q)
```

Examples

Remove named transactions:

```
$ svnadmin rmtxns /usr/local/svn/repos/ 1w 1x
```

Fortunately, the output of **lstxns** works great as the input for **rmtxns**:

```
$ svnadmin rmtxns /usr/local/svn/repos/  `svnadmin lstxns /usr/local/svn/repos/`
```

Which will remove all uncommitted transactions from your repository.

Name

svnadmin setlog -- Set the log-message on a revision.

Synopsis

```
svnadmin setlog REPOS_PATH -r REVISION FILE
```

Description

Set the log-message on revision REVISION to the contents of FILE.

This is similar to using **svn propset --revprop** to set the `svn:log` property on a revision, except that you can also use the option `--bypass-hooks` to avoid running any pre- or post-commit hooks, which is useful if the modification of revision properties has not been enabled in the pre-revprop-change hook.

Warning

Revision properties are not under version control, so this command will permanently overwrite the previous log message.

Switches

```
--revision (-r) ARG
--bypass-hooks
```

Examples

Set the log message for revision 19 to the contents of the file `msg`:

```
$ svnadmin setlog /usr/local/svn/repos/ -r 19 msg
```

Name

svnadmin verify -- Verify the data stored in the repository.

Synopsis

```
svnadmin verify REPOS_PATH
```

Description

Run this command if you wish to verify the integrity of your repository. This basically iterates through all revisions in the repository by internally dumping all revisions and discarding the output.

Examples

Verify a hung repository:

```
$ svnadmin verify /usr/local/svn/repos/
* Verified revision 1729.
```

svnlook

svnlook is a command-line utility for examining different aspects of a Subversion repository. It does not make any changes to the repository—it's just used for “peeking”. **svnlook** is typically used by the repository hooks, but a repository administrator might find it useful for diagnostic purposes.

Since **svnlook** works via direct repository access (and thus can only be used on the machine that holds the repository), it refers to the repository with a path, not a URL.

If no revision or transaction is specified, **svnlook** defaults to the youngest (most recent) revision of the repository.

svnlook Switches

Switches in **svnlook** are global, just like in **svn** and **svnadmin**; however, most switches only apply to one subcommand since the functionality of **svnlook** is (intentionally) limited in scope.

`--no-diff-deleted`
Prevents **svnlook** from printing differences for deleted files. The default behavior when a file is deleted in a transaction/revision is to print the same differences that you would see if you had left the file but removed all the content.

`--revision` (`-r`)
Specify a particular revision number that you wish to examine.

`--revprop`
Operates on a revision property instead of a Subversion property specific to a file or directory. This switch requires that you also pass a revision with the `--revision` (`-r`) switch. See the section called “Unversioned Properties” for more details on unversioned properties.

`--transaction` (`-t`)

Specify a particular transaction ID that you wish to examine.

`--show-ids`

Show the filesystem node revision IDs for each path in the filesystem tree.

svnlook

Name

svnlook author -- Print the author.

Synopsis

```
svnlook author REPOS_PATH
```

Description

Print the author of a revision or transaction in the repository.

Switches

```
--revision (-r)
--transaction (-t)
```

Examples

svnlook author is handy, but not very exciting:

```
$ svnlook author -r 40 /usr/local/svn/repos
sally
```

Name

svnlook cat -- Print the contents of a file.

Synopsis

```
svnlook cat REPOS_PATH PATH_IN_REPOS
```

Description

Print the contents of a file.

Switches

```
--revision (-r)
--transaction (-t)
```

Examples

This shows the contents of a file in transaction `ax8`, located at `/trunk/README`:

```
$ svnlook cat -t ax8 /usr/local/svn/repos /trunk/README

               Subversion, a version control system.
               =====================================

$LastChangedDate: 2003-07-17 10:45:25 -0500 (Thu, 17 Jul 2003) $

Contents:

     I. A FEW POINTERS
    II. DOCUMENTATION
   III. PARTICIPATING IN THE SUBVERSION COMMUNITY
…
```

Name

svnlook changed -- Print the paths that were changed.

Synopsis

```
svnlook changed REPOS_PATH
```

Description

Print the paths that were changed in a particular revision or transaction, as well as “svn update-style” status letters in the first two columns:

`'A '`
Item added to repository.

`'D '`
Item deleted from repository.

`'U '`
File contents changed.

`'_U'`
Properties of item changed.

`'UU'`
File contents and properties changed.

Files and directories can be distinguished, as directory paths are displayed with a trailing `'/'` character.

Switches

```
--revision (-r)
--transaction (-t)
```

Examples

This shows a list of all the changed files in revision 39 of a test repository:

```
$ svnlook changed -r 39 /usr/local/svn/repos
A   trunk/vendors/deli/
A   trunk/vendors/deli/chips.txt
A   trunk/vendors/deli/sandwich.txt
A   trunk/vendors/deli/pickle.txt
U   trunk/vendors/baker/bagel.txt
_U  trunk/vendors/baker/croissant.txt
UU  trunk/vendors/baker/pretzel.txt
D   trunk/vendors/baker/baguette.txt
```

Name

svnlook date -- Print the datestamp.

Synopsis

```
svnlook date REPOS_PATH
```

Description

Print the datestamp of a revision or transaction in a repository.

Switches

```
--revision (-r)
--transaction (-t)
```

Examples

This shows the date of revision 40 of a test repository:

```
$ svnlook date -r 40 /tmp/repos/
2003-02-22 17:44:49 -0600 (Sat, 22 Feb 2003)
```

Name

svnlook diff -- Print differences of changed files and properties.

Synopsis

```
svnlook diff REPOS_PATH
```

Description

Print GNU-style differences of changed files and properties in a repository.

Switches

```
--revision (-r)
--transaction (-t)
--no-diff-added
--no-diff-deleted
```

Examples

This shows a newly added (empty) file, a deleted file, and a copied file:

```
$ svnlook diff -r 40 /usr/local/svn/repos/
Copied: egg.txt (from rev 39, trunk/vendors/deli/pickle.txt)

Added: trunk/vendors/deli/soda.txt
==============================================================================

Modified: trunk/vendors/deli/sandwich.txt
==============================================================================
--- trunk/vendors/deli/sandwich.txt (original)
+++ trunk/vendors/deli/sandwich.txt 2003-02-22 17:45:04.000000000 -0600
@@ -0,0 +1 @@
+Don't forget the mayo!

Modified: trunk/vendors/deli/logo.jpg
==============================================================================
(Binary files differ)

Deleted: trunk/vendors/deli/chips.txt
==============================================================================

Deleted: trunk/vendors/deli/pickle.txt
==============================================================================
```

If a file has a non-textual `svn:mime-type` property, then the differences are not explicitly shown.

Name

svnlook dirs-changed -- Print the directories that were themselves changed.

Synopsis

```
svnlook dirs-changed REPOS_PATH
```

Description

Print the directories that were themselves changed (property edits) or whose file children were changed.

Switches

```
--revision (-r)
--transaction (-t)
```

Examples

This shows the directories that changed in revision 40 in our sample repository:

```
$ svnlook dirs-changed -r 40 /usr/local/svn/repos
trunk/vendors/deli/
```

Name

svnlook help

Synopsis

`Also svnlook -h and svnlook -?.`

Description

Displays the help message for svnlook. This command, like its brother **svn help**, is also your friend, even though you never call it anymore and forgot to invite it to your last party.

Alternate Names

?, h

Name

svnlook history -- Print information about the history of a path in the repository (or the root directory if no path is supplied).

Synopsis

```
svnlook history REPOS_PATH [PATH_IN_REPOS]
```

Description

Print information about the history of a path in the repository (or the root directory if no path is supplied).

Switches

```
--revision (-r)
--show-ids
```

Examples

This shows the history output for the path `/tags/1.0` as of revision 20 in our sample repository.

```
$ svnlook history -r 20 /usr/local/svn/repos /tags/1.0 --show-ids
REVISION   PATH <ID>
--------   ---------
      19   /tags/1.0 <1.2.12>
      17   /branches/1.0-rc2 <1.1.10>
      16   /branches/1.0-rc2 <1.1.x>
      14   /trunk <1.0.q>
      13   /trunk <1.0.o>
      11   /trunk <1.0.k>
       9   /trunk <1.0.g>
       8   /trunk <1.0.e>
       7   /trunk <1.0.b>
       6   /trunk <1.0.9>
       5   /trunk <1.0.7>
       4   /trunk <1.0.6>
       2   /trunk <1.0.3>
       1   /trunk <1.0.2>
```

Name

svnlook info -- Print the author, datestamp, log message size, and log message.

Synopsis

```
svnlook info REPOS_PATH
```

Description

Print the author, datestamp, log message size, and log message.

Switches

```
--revision (-r)
--transaction (-t)
```

Examples

This shows the info output for revision 40 in our sample repository.

```
$ svnlook info -r 40 /usr/local/svn/repos
sally
2003-02-22 17:44:49 -0600 (Sat, 22 Feb 2003)
15
Rearrange lunch.
```

Name

svnlook lock -- If a lock exists on a path in the repository, describe it.

Synopsis

```
svnlook lock REPOS_PATH PATH_IN_REPOS
```

Description

Print all information available for the lock at *`PATH_IN_REPOS`*. If *`PATH_IN_REPOS`* is not locked, print nothing.

Switches

None

Examples

This describes the lock on the file `tree.jpg`.

```
$ svnlook lock /svn/repos tree.jpg
UUID Token: opaquelocktoken:ab00ddf0-6afb-0310-9cd0-dda813329753
Owner: harry
Created: 2005-07-08 17:27:36 -0500 (Fri, 08 Jul 2005)
Expires:
Comment (1 line):
Rework the uppermost branches on the bald cypress in the foreground.
```

Name

svnlook log -- Print the log message.

Synopsis

```
svnlook log REPOS_PATH
```

Description

Print the log message.

Switches

```
--revision (-r)
--transaction (-t)
```

Examples

This shows the log output for revision 40 in our sample repository:

```
$ svnlook log /tmp/repos/
Rearrange lunch.
```

Name

svnlook propget -- Print the raw value of a property on a path in the repository.

Synopsis

```
svnlook propget REPOS_PATH PROPNAME [PATH_IN_REPOS]
```

Description

List the value of a property on a path in the repository.

Alternate Names

pg, pget

Switches

```
--revision (-r)
--transaction (-t)
--revprop
```

Examples

This shows the value of the “seasonings” property on the file `/trunk/sandwich` in the `HEAD` revision:

```
$ svnlook pg /usr/local/svn/repos seasonings /trunk/sandwich
mustard
```

Name

svnlook proplist -- Print the names and values of versioned file and directory properties.

Synopsis

```
svnlook proplist REPOS_PATH [PATH_IN_REPOS]
```

Description

List the properties of a path in the repository. With `--verbose`, show the property values too.

Alternate Names

pl, plist

Switches

```
--revision (-r)
--transaction (-t)
--verbose (-v)
--revprop
```

Examples

This shows the names of properties set on the file `/trunk/README` in the `HEAD` revision:

```
$ svnlook proplist /usr/local/svn/repos /trunk/README
  original-author
  svn:mime-type
```

This is the same command as in the previous example, but this time showing the property values as well:

```
$ svnlook --verbose proplist /usr/local/svn/repos /trunk/README
  original-author : fitz
  svn:mime-type : text/plain
```

Name

svnlook tree -- Print the tree.

Synopsis

```
svnlook tree REPOS_PATH [PATH_IN_REPOS]
```

Description

Print the tree, starting at *PATH_IN_REPOS* (if supplied, at the root of the tree otherwise), optionally showing node revision IDs.

Switches

```
--revision (-r)
--transaction (-t)
--show-ids
```

Examples

This shows the tree output (with node-IDs) for revision 40 in our sample repository:

```
$ svnlook tree -r 40 /usr/local/svn/repos --show-ids
/ <0.0.2j>
 trunk/ <p.0.2j>
  vendors/ <q.0.2j>
   deli/ <1g.0.2j>
    egg.txt <1i.e.2j>
    soda.txt <1k.0.2j>
    sandwich.txt <1j.0.2j>
```

Name

svnlook uuid -- Print the repository's `UUID`.

Synopsis

```
svnlook uuid REPOS_PATH
```

Description

Print the `UUID` for the repository. the UUID is the repository's *u*niversal *u*nique *id*entifier. The Subversion client uses this identifier to differentiate between one repository and another.

Examples

```
$ svnlook uuid /usr/local/svn/repos
e7fe1b91-8cd5-0310-98dd-2f12e793c5e8
```

Name

svnlook youngest -- Print the youngest revision number.

Synopsis

```
svnlook youngest REPOS_PATH
```

Description

Print the youngest revision number of a repository.

Examples

This shows the youngest revision of our sample repository:

```
$ svnlook youngest /tmp/repos/
42
```

svnserve

svnserve allows access to Subversion repositories using the `svn` network protocol. You can run svnserve either as a standalone server process, or you can have another process, such as **inetd**, **xinetd** or **sshd**, launch it for you.

Once the client has selected a repository by transmitting its URL, **svnserve** reads a file named `conf/svnserve.conf` in the repository directory to determine repository-specific settings such as what authentication database to use and what authorization policies to apply. See the section called “svnserve, a custom server” for details of the `svnserve.conf` file.

svnserve Switches

Unlike the previous commands we've described. **svnserve** has no subcommands—**svnserve** is controlled exclusively by switches.

`--daemon` (`-d`)
Causes **svnserve** to run in daemon mode. **svnserve** backgrounds itself and accepts and serves TCP/IP connections on the svn port (3690, by default).

`--listen-port=`*`PORT`*
Causes svnserve to listen on *`PORT`* when run in daemon mode.

`--listen-host=`*`HOST`*
Causes **svnserve** to listen on the interface specified by *`HOST`*, which may be either a hostname or an IP address.

`--foreground`
When used together with `-d`, this switch causes **svnserve** to stay in the foreground. This switch is mainly useful for debugging.

`--inetd` (`-i`)
Causes **svnserve** to use the stdin/stdout file descriptors, as is appropriate for a daemon running out of **inetd**.

`--help` (`-h`)
Displays a usage summary and exits.

`--version`
Displays version information, a list of repository back-end modules available, and exits.

`--root=`*`ROOT`* (`-r=`*`ROOT`*)
Sets the virtual root for repositories served by **svnserve**. The pathname in URLs provided by the client will be interpreted relative to this root, and will not be allowed to escape this root.

`--tunnel` (`-t`)
Causes **svnserve** to run in tunnel mode, which is just like the **inetd** mode of operation (serve one connection over stdin/stdout) except that the connection is considered to be pre-authenticated with the username of the current uid. This flag is selected by the client when running over a tunnel agent such as **ssh**.

`--tunnel-user NAME`
Used in conjunction with `--tunnel` switch; tells svnserve to assume that *`NAME`* is the authenticated user, rather than the UID of the svnserve process. Useful for users wishing to share a single system account over SSH, but maintaining separate commit identities.

`--threads` (`-T`)
When running in daemon mode, causes **svnserve** to spawn a thread instead of a process for each connection. The **svnserve** process still backgrounds itself at startup time.

`--listen-once` (`-X`)
Causes **svnserve** to accept one connection on the svn port, serve it, and exit. This option is mainly useful for debugging.

svnversion

Name

svnversion -- Summarize the local revision(s) of a working copy.

Synopsis

```
svnversion [OPTIONS] WC_PATH [TRAIL_URL]
```

Description

svnversion is a program for summarizing the revision mixture of a working copy. The resultant revision number, or revision range, is written to standard output.

`TRAIL_URL`, if present, is the trailing portion of the URL used to determine if *`WC_PATH`* itself is switched (detection of switches within *`WC_PATH`* does not rely on *`TRAIL_URL`*).

Switches

Like **svnserve**, **svnversion** has no subcommands, it only has switches.

`--no-newline` (`-n`)
Omit the usual trailing newline from the output.

`--committed` (`-c`)
Use the last-changed revisions rather than the current (i.e., highest locally available) revisions.

`--help` (`-h`)
Print a help summary.

`--version`
Print the version of **svnversion** and exit with no error.

Examples

If the working copy is all at the same revision (for example, immediately after an update), then that revision is printed out:

```
$ svnversion .
4168
```

You can add TRAIL_URL to show that the working copy is not switched from what you expect:

```
$ svnversion . /repos/svn/trunk
4168
```

For a mixed-revision working copy, the range of revisions present is printed:

```
$ svnversion .
4123:4168
```

If the working copy contains modifications, a trailing "M" is added:

```
$ svnversion .
4168M
```

If the working copy is switched, a trailing "S" is added:

```
$ svnversion .
4168S
```

Thus, here is a mixed-revision, switched working copy containing some local modifications:

```
$ svnversion .
4212:4168MS
```

If invoked on a directory that is not a working copy, **svnversion** assumes it is an exported working copy and prints "exported":

```
$ svnversion .
exported
```

mod_dav_svn

Name

`mod_dav_svn` Configuration Directives -- Apache configuration directives for serving Subversion repositories through Apache HTTP Server.

Description

This section briefly describes each of the Subversion Apache configuration directives. For an in-depth description of configuring Apache with Subversion, see the section called “httpd, the Apache HTTP server”.)

Directives

`DAV svn`
This directive must be included in any `Directory` or `Location` block for a Subversion repository. It tells httpd to use the Subversion backend for mod_dav to handle all requests.

`SVNAutoversioning On`
This directive allows write requests from WebDAV clients to result in automatic commits. A generic log message is auto-generated and attached to each revision. If you enable Autoversioning, you'll likely want to set `ModMimeUsePathInfo On` so that `mod_mime` can set `svn:mime-type` to the correct mime-type automatically (as best as `mod_mime` is able to, of course). For more information, see Appendix B, *WebDAV and Autoversioning*

`SVNPath`
This directive specifies the location in the filesystem for a Subversion repository's files. In a configuration block for a Subversion repository, either this directive or `SVNParentPath` must be present, but not both.

`SVNSpecialURI`
Specifies the URI component (namespace) for special Subversion resources. The default is “`!svn`”, and most administrators will never use this directive. Only set this if there is a pressing need to have a file named `!svn` in your repository. If you change this on a server already in use, it will break all of the outstanding working copies and your users will hunt you down with pitchforks and flaming torches.

`SVNReposName`
Specifies the name of a Subversion repository for use in `HTTP GET` requests. This value will be prepended to the title of all directory listings (which are served when you navigate to a Subversion repository with a web browser). This directive is optional.

`SVNIndexXSLT`
Specifies the URI of an XSL transformation for directory indexes. This directive is optional.

`SVNParentPath`
Specifies the location in the filesystem of a parent directory whose child directories are Subversion repositories. In a configuration block for a Subversion repository, either this directive or `SVNPath` must be present, but not both.

`SVNPathAuthz`
Control path-based authorization by enabling or disabling subrequests. See the section called “Disabling Path-based Checks” for details.

Subversion properties

Name

Subversion-defined properties -- properties defined by Subversion to control behavior.

Description

Subversion allows users to invent arbitrarily-named versioned properties on files and directories, as well as unversioned properties on revisions. The only restriction is on properties prefixed with “svn:”. Properties in that namespace are reserved for Subversion's own use. While these properties may be set by users to control Subversion's behavior, users may not invent new “svn:” properties.

Versioned Properties

`svn:executable`
If present on a file, the client will make the file executable in Unix-hosted working copies. See the section called “`svn:executable`”.

`svn:mime-type`
If present on a file, the value indicates the file's mime-type. This allows the client to decide whether line-based contextual merging is safe to perform during updates, and can also affect how the file behaves when fetched via web browser. See the section called “`svn:mime-type`”.

`svn:ignore`
If present on a directory, the value is a list of unversioned file patterns to be ignored by **svn status** and other subcommands. See the section called “`svn:ignore`”

`svn:keywords`
If present on a file, the value tells the client how to expand particular keywords within the file. See the section called “`svn:keywords`”.

`svn:eol-style`
If present on a file, the value tells the client how to manipulate the file's line-endings in the working copy. See the section called “`svn:eol-style`”.

`svn:externals`
If present on a directory, the value is a multi-line list of other paths and URLs the client should check out. See the section called “`svn:externals`”.

`svn:special`
If present on a file, indicates that the file is not an ordinary file, but a symbolic link or other special object. See the section called “`svn:special`”.

`svn:needs-lock`

`svn:date`
Contains the UTC time the revision was created, in ISO format. The value comes from the server machine's clock.

`svn:log`
Contains the log message describing the revision.

`svn:autoversioned`
If present, the revision was created via the autoversioning feature. See the section called “Autoversioning”.

Appendix A. Subversion for CVS Users

This appendix is a guide for CVS users new to Subversion. It's essentially a list of differences between the two systems as “viewed from 10,000 feet”. For each section, we provide backreferences to relevant chapters when possible.

Although the goal of Subversion is to take over the current and future CVS user base, some new features and design changes were required to fix certain “broken” behaviors that CVS had. This means that, as a CVS user, you may need to break 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 stores its data in RCS files; 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.

For more details on this topic, see the section called “Revisions”.

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, as a former CVS user:

- The **svn add** and **svn delete** commands work on directories now, just as they work on files. So do **svn copy** and **svn move**. However, these commands do *not* cause any kind of immediate change in the repository. Instead, the working items are simply “scheduled” for addition or deletion. No repository changes happen until you run **svn commit**.
- Directories aren't dumb containers anymore; they have revision numbers like files. (Or more properly, it's correct to talk about “directory `foo/` in revision 5”.)

Let's talk more about that last point. Directory versioning is a hard problem; because we want to allow mixed-revision working copies, there are some limitations on how far we can abuse this model.

From a theoretical point of view, we define “revision 5 of directory `foo`” to mean a specific collection of directory-entries and properties. Now suppose we start adding and removing files from `foo`, and then commit. It would be a lie to say that we still have revision 5 of `foo`. However, if we bumped `foo`'s revision number after the commit, that would be a lie too; there may be other changes to `foo` we haven't yet received, because we haven't updated yet.

Subversion deals with this problem by quietly tracking committed adds and deletes in the `.svn` area.

When you eventually run **svn update**, all accounts are settled with the repository, and the directory's

svn status prints all files that have local modifications. By default, the repository is not contacted. While this subcommand accepts a fair number of options, the following are the most commonly used ones:

`-u`
Contact the repository to determine, and then display, out-of-dateness information.

`-v`
Show *all* entries under version control.

`-N`
Run non-recursively (do not descend into subdirectories).

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 `--show-updates` (`-u`) switch, a longer output format is used:

```
$ svn status -u
M            1047   foo.c
       *     1045   faces.html
       *            bloo.png
M            1050   bar/baz.c
Status against 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 absence of any revision number next to `bloo.png` means that it doesn't yet exist in the working copy.)

Lastly, here's a quick summary of the most common status codes that you may see:

```
A    Resource is scheduled for Addition
D    Resource is scheduled for Deletion
M    Resource has local Modifications
C    Resource has Conflicts (changes have not been completely merged
       between the repository and working copy version)
X    Resource is eXternal to this working copy (may come from another
       repository).  See the section called "svn:externals"
```

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.

For a more detailed discussion of **svn update**, see the section called “Update Your Working Copy”.

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 in Chapter 4, *Branching and Merging*.

Warning

Since Subversion treats branches and tags as ordinary directories, always remember to check out the `trunk` (`http://svn.example.com/repos/calc/trunk/`) of your project, and not the project itself (`http://svn.example.com/repos/calc/`). If you make the mistake of checking out the project itself, you'll wind up with a working copy that contains a copy of your project for every branch and tag you have.[1]

Metadata Properties

A new feature of Subversion is that you can attach arbitrary metadata (or “properties”) to files and directories. Properties are arbitrary name/value pairs associated with files and directories 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”.

Conflict Resolution

CVS marks conflicts with in-line “conflict markers”, and prints a `C` during an update. Historically, this has caused problems, because CVS isn't doing enough. 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 files containing conflict-markers.

Subversion solves this problem by making conflicts more tangible. It remembers that a file is in a state of conflict, and won't allow you to commit your changes until you run **svn resolved**. See the section called “Resolve Conflicts (Merging Others' Changes)” for more details.

Binary Files and Translation

In the most general sense, Subversion handles binary files more gracefully than CVS does. Because CVS uses RCS, it can only store successive full copies of a changing binary file. Subversion, however, expresses differences between files using a binary-differencing algorithm, regardless of whether they contain textual or binary data. That means that all files are stored differentially (compressed) in the repository.

CVS users have to mark binary files with `-kb` flags, to prevent data from being garbled (due to keyword

[1]That is, providing you don't run out of disk space before your checkout finishes.

expansion and line-ending translations). They sometimes forget to do this.

Subversion takes the more paranoid route—first, it never performs any kind of keyword or line-ending translation unless you explicitly ask it do so (see the section called “`svn:keywords`” and the section called “`svn:eol-style`” for more details). By default, Subversion treats all file data as literal byte strings, and files are always stored in the repository in an untranslated state.

Second, Subversion maintains an internal notion of whether a file is “text” or “binary” data, but this notion is *only* extant in the working copy. During an **svn update**, Subversion will perform contextual merges on locally modified text files, but will not attempt to do so for binary files.

To determine whether a contextual merge is possible, Subversion examines the `svn:mime-type` property. If the file has no `svn:mime-type` property, or has a mime-type that is textual (e.g. `text/*`), Subversion assumes it is text. Otherwise, Subversion assumes the file is binary. Subversion also helps users by running a binary-detection algorithm in the **svn import** and **svn add** commands. These commands will make a good guess and then (possibly) set a binary `svn:mime-type` property on the file being added. (If Subversion guesses wrong, the user can always remove or hand-edit the property.)

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 (e.g. adds or removes components), then a call to **svn update** will update the working copy appropriately, adding and removing components.

Subversion defines modules as a list of directories within a directory property: see the section called “Externals Definitions”.

Authentication

With CVS's pserver, you are required to “login” to the server before any read or write operation—you even have to login for anonymous operations. With a Subversion repository using Apache **httpd** or **svnserve** as the server, you don't provide any authentication credentials at the outset—if an operation that you perform requires authentication, the server will challenge you for your credentials (whether those credentials are username and password, a client certificate, or even both). So if your repository is world-readable, you will not be required to authenticate at all for read operations.

As with CVS, Subversion still caches your credentials on disk (in your `~/.subversion/auth/` directory) unless you tell it not to by using the `--no-auth-cache` switch.

The exception to this behavior, however, is in the case of accessing an **svnserve** server over an SSH tunnel, using the `svn+ssh://`

One such tool is cvs2svn (http://cvs2svn.tigris.org/), a Python script originally created by members of Subversion's own development community. Others include Lev Serebryakov's RefineCVS (http://lev.serebryakov.spb.ru/refinecvs/). These tools have various levels of completeness, and may make entirely different decisions about how to handle your CVS repository history. Whichever tool you decide to use, be sure to perform as much verification as you can stand on the conversion results—after all, you've worked hard to build that history!

For an updated collection of links to known converter tools, visit the Links page of the Subversion website (http://subversion.tigris.org/project_links.html).

Appendix B. WebDAV and Autoversioning

WebDAV is an extension to HTTP, and is growing more and more popular as a standard for file-sharing. Today's operating systems are becoming extremely Web-aware, and many now have built-in support for mounting “shares” exported by WebDAV servers.

If you use Apache/mod_dav_svn as your Subversion network server, then to some extent, you are also running a WebDAV server. This appendix gives some background on the nature of this protocol, how Subversion uses it, and how well Subversion interoperates with other software that is WebDAV-aware.

Basic WebDAV Concepts

This section provides a very brief, very general overview to the ideas behind WebDAV. It should lay the foundation for understanding WebDAV compatibility issues between clients and servers.

Original WebDAV

RFC 2518 defines a set of concepts and accompanying extension methods to HTTP 1.1 that make the web into a more universal read/write medium. The basic idea is that a WebDAV-compliant web server can act like a generic file server; clients can mount shared folders that behave much like NFS or SMB filesystems.

The tragedy, though, is that the RFC 2518 WebDAV specification does not provide any sort of model for version control, despite the “V” in DAV. Basic WebDAV clients and servers assume only one version of each file or directory exists, and can be repeatedly overwritten.

Here are the concepts and terms introduced in basic WebDAV:

Resources
WebDAV lingo refers to any server-side object (that can be described with a URI) as a *resource*.

New write methods
Beyond the standard HTTP `PUT` method (which creates or overwrites a web resource), WebDAV defines new `COPY` and `MOVE` methods for duplicating or rearranging resources.

Collections
A *collection* is the WebDAV term for a grouping of resources. In most cases, it is analogous to a directory. Whereas file resources can be written or created with a `PUT` method, collection resources are created with the new `MKCOL` method.

Properties
This is the same idea present in Subversion—metadata attached to files and collections. A client can list or retrieve properties attached to a resource with the new `PROPFIND` method, and can change them with the `PROPPATCH` method. Some properties are wholly created and controlled by users (e.g. a property called “color”), and others are wholly created and controlled by the WebDAV server (e.g. a property that contains the last modification time of a file). The former kind are called *dead properties*, and the latter kind are called *live properties*.

Locking
A WebDAV server may decide to offer a locking feature to clients—this part of the specification is optional, although most WebDAV servers do offer the feature. If present, then clients can use the new `LOCK` and `UNLOCK` methods to mediate access to a resource. In most cases these methods are used to create exclusive write locks (as discussed in the section called “The Lock-Modify-Unlock Solution”), although shared write locks are also

possible in some server implementations.

Access control
A more recent specification (RFC 3744) defines a system for defining access control lists (ACLs) on WebDAV resources. Some clients and servers have begun to implement this feature.

DeltaV Extensions

Because RFC 2518 left out versioning concepts, another committee was left with the responsibility of writing RFC 3253, which adds versioning to WebDAV, a.k.a. “DeltaV”. WebDAV/DeltaV clients and servers are often called just “DeltaV” programs, since DeltaV implies the existence of basic WebDAV.

DeltaV introduces a whole slew of new acronyms, but don't be intimidated. The ideas are fairly straightforward:

Per-resource versioning
Like CVS and other version-control systems, DeltaV assumes that each resource has a potentially infinite number of states. A client begins by placing a resource under version control using the new `VERSION-CONTROL` method.

Server-side working-copy model
Some DeltaV servers support the ability to create a virtual workspace on the server, where all of your work is performed. Clients use the `MKWORKSPACE` method to create a private area, then indicate they want to change specific resources by “checking them out” into the workspace, editing them, and “checking them in” again. In HTTP terms, the sequence of methods would be `CHECKOUT`, `PUT`, `CHECKIN`.

Client-side working-copy model
Some DeltaV servers also support the idea that the client may have a private working copy on local disk. When the client wants to commit changes to the server, it begins by creating a temporary server transaction (called an *activity*) with the `MKACTIVITY` method. The client then performs a `CHECKOUT` on each resource it wishes to change and sends `PUT` requests. Finally, the client performs a `CHECKIN` resource, or sends a `MERGE` request to check in all resources at once.

Configurations
DeltaV allows you define flexible collections of resources called “configurations”, which don't necessarily correspond to particular directories. A configuration can be made to point to specific versions of files, and then a “baseline” snapshot can be made, much like a tag.

Extensibility
DeltaV defines a new method, `REPORT`, which allows the client and server to perform customized data exchanges. While DeltaV defines a number of standardized history reports that a client can request, the server is also free to define custom reports. The client sends a `REPORT` request with a properly-labeled XML body full of custom data; assuming the server understands the specific report-type, it responds with an equally custom XML body. This technique is very similar to XML-RPC.

Subversion and DeltaV

The original WebDAV standard has been widely successful. Every modern computer operating system has a general WebDAV client built-in (details to follow), and a number of popular standalone applications are also able to speak WebDAV — Microsoft Office, Dreamweaver, and Photoshop to name a few. On the server end, the Apache webserver has been able to provide WebDAV services since 1998 and is considered the de-facto open-source standard. There are several other commercial WebDAV servers available, including Microsoft's own IIS.

DeltaV, unfortunately, has not been so successful. It's very difficult to find any DeltaV clients or servers.

The few that do exist are relatively unknown commercial products, and thus it's very difficult to test interoperability. It's not entirely clear as to why DeltaV has remained stagnant. Some argue that the specification is just too complex, others argue that while WebDAV's features have mass appeal (even the least technical users appreciate network file-sharing), version control features aren't interesting or necessary for most users. Finally, some have argued that DeltaV remains unpopular because there's still no open-source server product which implements it.

When Subversion was still in its design phase, it seemed like a great idea to use Apache httpd as the main network server. It already had a module to provide WebDAV services. DeltaV was a relatively new specification. The hope was that the Subversion server module (mod_dav_svn) would eventually evolve into an open-source DeltaV reference implementation. Unfortunately, DeltaV has a very specific versioning model that doesn't quite line up with Subversion's model. Some concepts were mappable, others were not.

The upshot is that

1. The Subversion client is not a fully-implemented DeltaV client.

   The client needs certain things from the server that DeltaV cannot provide, and thus is largely dependent on a number of Subversion-specific `REPORT` requests that only mod_dav_svn understands.

2. mod_dav_svn is not a fully-implemented DeltaV server.

   Many portions of the DeltaV specification were irrelevant to Subversion, and thus left unimplemented.

There is still some debate in the developer community as to whether or not it's worthwhile to remedy either of these situations. It's fairly unrealistic to change Subversion's design to match DeltaV, so there's probably no way the client can ever learn to get everything it needs from a general DeltaV server. On the other hand, mod_dav_svn *could* be further developed to implement all of DeltaV, but it's hard to find motivation to do so—there are almost no DeltaV clients to interoperate with.

Autoversioning

While the Subversion client is not a full DeltaV client, nor the Subversion server a full DeltaV server, there's still a glimmer of WebDAV interoperability to be happy about: it's called autoversioning.

Autoversioning is an optional feature defined in the DeltaV standard. A typical DeltaV server will reject an ignorant WebDAV client attempting to do a `PUT` to a file that's under version control. To change a version-controlled file, the server expects a series proper versioning requests: something like `MKACTIVITY`, `CHECKOUT`, `PUT`, `CHECKIN`. But if the DeltaV server supports autoversioning, then write-requests from basic WebDAV clients are accepted. The server behaves as if the client had issued the proper series of versioning requests, performing a commit under the hood. In other words, it allows a DeltaV server to interoperate with ordinary WebDAV clients that don't understand versioning.

Because so many operating systems already have integrated WebDAV clients, the use case for this feature borders on fantastical: imagine an office of ordinary users running Microsoft Windows or Mac OS. Each user “mounts” the Subversion repository, which appears to be an ordinary network folder. They use the shared folder as they always do: open files, edit them, save them. Meanwhile, the server is automatically versioning everything. Any administrator (or knowledgeable user) can still use a Subversion client to search history and retrieve older versions of data.

This scenario isn't fiction: it's real and it works, as of Subversion 1.2 and later. To activate autoversioning in mod_dav_svn, use the `SVNAutoversioning` directive within the `httpd.conf` Location block, like so:

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

When SVNAutoversioning is active, write requests from WebDAV clients result in automatic commits. A generic log message is auto-generated and attached to each revision.

Before activating this feature, however, understand what you're getting into. WebDAV clients tend to do *many* write requests, resulting in a huge number of automatically committed revisions. For example, when saving data, many clients will do a `PUT` of a 0-byte file (as a way of reserving a name) followed by another `PUT` with the real filedata. The single file-write results in two separate commits. Also consider that many applications auto-save every few minutes, resulting in even more commits.

If you have a post-commit hook program that sends email, you may want to disable email generation either altogether, or on certain sections of the repository; it depends on whether you think the influx of emails will still prove to be valuable notifications or not. Also, a smart post-commit hook program can distinguish between a transaction created via autoversioning and one created through a normal **svn commit**. The trick is to look for a revision property named `svn:autoversioned`. If present, the commit was made by a generic WebDAV client.

Another feature that may be a useful complement for `SVNAutoversioning` comes from Apache's `mod_mime` module. If a generic WebDAV client adds a new file to the repository, there's no opportunity for the user to set the the `svn:mime-type` property. This might cause the file to appear as “generic” icon when viewed within a WebDAV shared folder, not having an association with any application. One remedy is to have a sysadmin (or other Subversion-knowledgable person) check out a working copy and manually set the `svn:mime-type` property on necessary files. But there's potentially no end to such cleanup tasks. Instead, you can use the `ModMimeUsePathInfo` directive in your Subversion `<Location>` block:

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

  ModMimeUsePathInfo on

</Location>
```

This directive allows `mod_mime` to attempt automatic deduction of the mime-type on new files that enter the repository via autoversioning. The module looks at the file's named extension and possibly the contents as well; if the file matches some common patterns, then the the file's `svn;mime-type` property will be set automatically.

Client Interoperability

All WebDAV clients fall into one of three categories—standalone applications, file-explorer extensions, or filesystem implementations. These categories broadly define the types of WebDAV functionality available to users. Table B.1, “Common WebDAV Clients” gives our categorization and a quick description of some common pieces of WebDAV-enabled software. More details about these software offerings, as well as their general category, can be found in the sections that follow.

Table B.1. Common WebDAV Clients

Software	Category	Description
Adobe Photoshop	Standalone WebDAV applications	Image editing software, allowing direct opening from, and writing to, WebDAV URLs
Cadaver	Standalone WebDAV applications	Command-line WebDAV client supporting file transfer, tree, and locking operations
DAV Explorer	Standalone WebDAV applications	GUI tool for exploring WebDAV shares
davfs2	WebDAV filesystem implementation	Linux file system driver that allows you to mount a WebDAV share
GNOME Nautilus	File-explorer WebDAV extensions	GUI file explorer able to perform tree operations on a WebDAV share
KDE Konqueror	File-explorer WebDAV extensions	GUI file explorer able to perform tree operations on a WebDAV share
Mac OS X	WebDAV filesystem implementation	Operating system with built-in support for mounting WebDAV shares locally
Macromedia Dreamweaver	Standalone WebDAV applications	Web production software able to directly read from and write to WebDAV URLs
Microsoft Office	Standalone WebDAV applications	Office productivity suite with several components able to directly read from and write to WebDAV URLs
Microsoft Web Folders	File-explorer WebDAV extensions	GUI file explorer program able to perform tree operations on a WebDAV share
Novell NetDrive	WebDAV filesystem implementation	Drive-mapping program for assigning Windows drive letters to a mounted remote WebDAV share
SRT WebDrive	WebDAV filesystem implementation	File transfer software which, among other things, allows the assignment of Windows drive letters to a mounted remote WebDAV share

Standalone WebDAV applications

A WebDAV application is a program which contains built-in functionality for speaking WebDAV protocols with a WebDAV server. We'll cover some of the most popular programs with this kind of WebDAV support.

Microsoft Office, Dreamweaver, Photoshop

On Windows, there are several well-known applications that contain integrated WebDAV client functionality, such as Microsoft's Office, [1] Adobe's Photoshop, and Macromedia's Dreamweaver programs. They're able to directly open and save to URLs, and tend to make heavy use of WebDAV locks when editing a file.

Note that while many of these programs also exist for the Mac OS X, they do not appear to support WebDAV directly on that platform. In fact, on Mac OS X, the File->Open dialog box doesn't allow one

[1] WebDAV support was removed from Microsoft Access for some reason, but exists in the rest of the Office suite.

to type a path or URL at all. It's likely that the WebDAV features were deliberately left out of Macintosh versions of these programs, since OS X already provides such excellent low-level filesystem support for WebDAV.

Cadaver, DAV Explorer

Cadaver is a bare-bones Unix commandline program for browsing and changing WebDAV shares. Like the Subversion client, it uses the neon HTTP library—not surprisingly, both neon and cadaver are written by the same author. Cadaver is free software (GPL license) and is available at http://www.webdav.org/cadaver/.

Using cadaver is similar to using a commandline FTP program, and thus it's extremely useful for basic WebDAV debugging. It can be used to upload or download files in a pinch, and also to examine properties, copy, move, lock or unlock files:

```
$ cadaver http://host/repos
dav:/repos/> ls
Listing collection `/repos/': succeeded.
Coll: > foobar                                 0  May 10 16:19
```

filesystems. It works well enough. In Windows 98, the feature might need to be explicitly installed if Web Folders aren't already visible inside “My Computer”. In Windows 2000, simply add a new “network place”, enter the URL, and the WebDAV share will pop up for browsing.

With the release of Windows XP, Microsoft started shipping a new implementation of Web Folders, known as the “WebDAV mini-redirector”. The new implementation is a filesystem-level client, allowing WebDAV shares to be mounted as drive letters. Unfortunately, this implementation is incredibly buggy. The client usually tries to convert http URLs (`http://host/repos`) into UNC share notation (`\\host\repos`); it also often tries to use Windows Domain authentication to respond to basic-auth HTTP challenges, sending usernames as `HOST\username`. These interoperability problems are severe and documented in numerous places around the web, to the frustration of many users. Even Greg Stein, the original author of Apache's WebDAV module, recommends against trying to use XP Web Folders against an Apache server.

It turns out that the original “Explorer-only” Web Folders implementation isn't dead in XP, it's just buried. It's still possible to find it by using this technique:

1. Go to 'Network Places'.

2. Add a new network place.

3. When prompted, enter the URL of the repository, but *include a port number* in the URL. For example, `http://host/repos` would be entered as `http://host:80/repos` instead.

4. Respond to any authentication prompts.

There are a number of other rumored workarounds to the problems, but none of them seem to work on all versions and patchlevels of Windows XP. In our tests, only the previous algorithm seems to work consistently on every system. The general consensus of the WebDAV community is that you should avoid the new Web Folders implementation and use the old one instead, and that if you need real a real filesystem-level client for Windows XP, then use a third-party program like WebDrive or NetDrive.

A final tip: if you're attempting to use XP Web Folders, make sure you have the absolute latest version as a low-level filesystem module, typically within the operating system's kernel. This means that the DAV share is mounted like any other network filesystem, similar to mounting an NFS share on Unix, or

[2] From the Darwin terminal, one can also run `mount -t webdav URL /mountpoint`

Appendix C. Third Party Tools

Subversion's modular design (covered in the section called “Layered Library Design”) and the availability of language bindings (as described in the section called “Using Languages Other than C and C++”) make it a likely candidate for use as an extension or backend to other pieces of software. For a listing of many third-party tools that are using Subversion functionality under-the-hood, check out the Links page on the Subversion website (http://subversion.tigris.org/project_links.html).

Appendix D. Copyright

A summary of the license is given below, followed by the full legal
text.

You are free:

* to copy, distribute, display, and perform the work
* to make derivative works
* to make commercial use of the work

Under the following conditions:

Attribution. You must give the original author credit.

* For any reuse or distribution, you must make clear to others the license terms of this work.

* Any of these conditions can be waived if you get permission from the author.

Your fair use and other rights are in no way affected by the above.

The above is a summary of the full license below.

==

Creative Commons Legal Code
Attribution 2.0

CREATIVE COMMONS CORPORATION IS NOT A LAW FIRM AND DOES NOT PROVIDE
LEGAL SERVICES. DISTRIBUTION OF THIS LICENSE DOES NOT CREATE AN
ATTORNEY-CLIENT RELATIONSHIP. CREATIVE COMMONS PROVIDES THIS
INFORMATION ON AN "AS-IS" BASIS. CREATIVE COMMONS MAKES NO WARRANTIES
REGARDING THE INFORMATION PROVIDED, AND DISCLAIMS LIABILITY FOR
DAMAGES RESULTING FROM ITS USE.

License

THE WORK (AS DEFINED BELOW) IS PROVIDED UNDER THE TERMS OF THIS
CREATIVE COMMONS PUBLIC LICENSE ("CCPL" OR "LICENSE"). THE WORK IS
PROTECTED BY COPYRIGHT AND/OR OTHER APPLICABLE LAW. ANY USE OF THE
WORK OTHER THAN AS AUTHORIZED UNDER THIS LICENSE OR COPYRIGHT LAW IS
PROHIBITED.

BY EXERCISING ANY RIGHTS TO THE WORK PROVIDED HERE, YOU ACCEPT AND
AGREE TO BE BOUND BY THE TERMS OF THIS LICENSE. THE LICENSOR GRANTS
YOU THE RIGHTS CONTAINED HERE IN CONSIDERATION OF YOUR ACCEPTANCE OF
SUCH TERMS AND CONDITIONS.

e.

For the avoidance of doubt, where the work is a musical composition:

i. Performance Royalties Under Blanket Licenses. Licensor waives the exclusive right to collect, whether individually or via a performance rights society (e.g. ASCAP, BMI, SESAC), royalties for the public performance or public digital performance (e.g. webcast) of the Work.

ii. Mechanical Rights and Statutory Royalties. Licensor waives the exclusive right to collect, whether individually or via a music rights agency or designated agent (e.g. Harry Fox Agency), royalties for any phonorecord You create from the Work ("cover version") and distribute, subject to the compulsory license created by 17 USC Section 115 of the US Copyright Act (or the equivalent in other jurisdictions).

f. Webcasting Rights and Statutory Royalties. For the avoidance of doubt, where the Work is a sound recording, Licensor waives the exclusive right to collect, whether individually or via a performance-rights society (e.g. SoundExchange), royalties for the public digital performance (e.g. webcast) of the Work, subject to the compulsory license created by 17 USC Section 114 of the US Copyright Act (or the equivalent in other jurisdictions).

The above rights may be exercised in all media and formats whether now known or hereafter devised. The above rights include the right to make such modifications as are technically necessary to exercise the rights in other media and formats. All rights not expressly granted by Licensor are hereby reserved.

4. Restrictions.The license granted in Section 3 above is expressly made subject to and limited by the following restrictions:

a. You may distribute, publicly display, publicly perform, or publicly digitally perform the Work only under the terms of this License, and You must include a copy of, or the Uniform Resource Identifier for, this License with every copy or phonorecord of the Work You distribute, publicly display, publicly perform, or publicly digitally perform. You may not offer or impose any terms on the Work that alter or restrict the terms of this License or the recipients' exercise of the rights granted hereunder. You may not sublicense the Work. You must keep intact all notices that refer to this License and to the disclaimer of warranties. You may not distribute, publicly display, publicly perform, or publicly digitally perform the Work with any technological measures that control access or use of the Work in a manner inconsistent with the terms of this License Agreement. The above applies to the Work as incorporated in a Collective Work, but this does not require the Collective Work apart from the Work itself to be made subject to the terms of this License. If You create a Collective Work, upon notice from any Licensor You must, to the extent practicable, remove from the Collective Work any reference to such v1ee 4tentthe Collective wit9ight

b. Each time You distribute or publicly digitally perform a Derivative Work, Licensor offers to the recipient a license to the original Work on the same terms and conditions as the