Subversion
subversion.tigris.org

Subversion 1.6

Version Control with Subversion

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

Linbrary™ - Linux Documentation Library
www.linbrary.com

Subversion 1.6

Official Guide

Version Control with Subversion

Fultus™ Books

Subversion 1.6 Official Guide

Version Control with Subversion

ISBN-10: 1-59682-169-8
ISBN-13: 978-1-59682-169-9

Cover design and book layout by Fultus Corporation

Published by Fultus Corporation

Publisher Web: *www.fultus.com*
Linbrary - Linux Library: *www.linbrary.com*
Online Bookstore: *store.fultus.com*
email: *production@fultus.com*

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 ask, but of the questions the FAQ's author *wishes* people would ask. 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, such as "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 60-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 decided to publish a full-length Subversion book, the path of least resistance was obvious: just expand the Subversion handbook.

The three coauthors 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 also 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 was 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 and still watching, and the authors 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 only 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, March 14, 2004.

Preface

"It is important not to let the perfect become the enemy of the good, even when you can agree on what perfect is. Doubly so when you can't. As unpleasant as it is to be trapped by past mistakes, you can't make any progress by being afraid of your own shadow during design."

--Greg Hudson, Subversion developer

In the world of open source software, the Concurrent Versions System (CVS) was the tool of choice for version control for many years. And rightly so. CVS was open source software itself, and its nonrestrictive modus operandi and support for networked operation allowed dozens of geographically dispersed programmers to share their work. It fit the collaborative nature of the open source world very well. CVS and its semi-chaotic development model have since become cornerstones of open source culture.

But CVS was not without its flaws, and simply fixing those flaws promised to be an enormous effort. Enter Subversion. Subversion was designed to be a successor to CVS, and its originators set out to win the hearts of CVS users in two ways—by creating an open source system with a design (and "look and feel") similar to CVS, and by attempting to avoid most of CVS's noticeable flaws. While the result isn't necessarily the next great evolution in version control design, Subversion *is* very powerful, very usable, and very flexible. And for the most part, almost all newly started open source projects now choose Subversion instead of CVS.

This book is written to document the 1.6 series of the Subversion version control system. We have made every attempt to be thorough in our coverage. However, Subversion has a thriving and energetic development community, so already a number of features and improvements are planned for future versions that may change some of the commands and specific notes in this book.

Audience

This book is written for computer-literate folk who want to use Subversion to manage their data. While Subversion runs on a number of different operating systems, its primary user interface is command-line-based. That command-line tool (***svn***), and some auxiliary programs, are the focus of this book.

For consistency, the examples in this book assume that the reader is using a Unix-like operating system and is relatively comfortable with Unix and command-line interfaces. That

said, the ***svn*** program also runs on non-Unix platforms such as Microsoft Windows. With a few minor exceptions, such as the use of backward slashes (\) instead of forward slashes (/) for path separators, the input to and output from this tool when run on Windows are identical to its Unix counterpart.

Most readers are probably programmers or system administrators who need to track changes to source code. This is the most common use for Subversion, and therefore it is the scenario underlying all of the book's examples. But Subversion can be used to manage changes to any sort of information—images, music, databases, documentation, and so on. To Subversion, all data is just data.

While this book is written with the assumption that the reader has never used a version control system, we've also tried to make it easy for users of CVS (and other systems) to make a painless leap into Subversion. Special sidebars may mention other version control systems from time to time, and Appendix B summarizes many of the differences between CVS and Subversion.

Note also that the source code examples used throughout the book are only examples. While they will compile with the proper compiler incantations, they are intended to illustrate a particular scenario and not necessarily to serve as examples of good programming style or practices.

How to Read This Book

Technical books always face a certain dilemma: whether to cater to *top-down* or to *bottom-up* learners. A top-down learner prefers to read or skim documentation, getting a large overview of how the system works; only then does she actually start using the software. A bottom-up learner is a "learn by doing" person—someone who just wants to dive into the software and figure it out as she goes, referring to book sections when necessary. Most books tend to be written for one type of person or the other, and this book is undoubtedly biased toward top-down learners. (And if you're actually reading this section, you're probably already a top-down learner yourself!) However, if you're a bottom-up person, don't despair. While the book may be laid out as a broad survey of Subversion topics, the content of each section tends to be heavy with specific examples that you can try-by-doing. For the impatient folks who just want to get going, you can jump right to *Appendix A*, "Subversion Quick-Start Guide" (page 439).

Regardless of your learning style, this book aims to be useful to people of widely different backgrounds—from those with no previous experience in version control to experienced system administrators. Depending on your own background, certain chapters may be more or less important to you. The following can be considered a "recommended reading list" for various types of readers:

Experienced system administrators

The assumption here is that you've probably used version control before and are dying to get a Subversion server up and running ASAP. *Chapter 5,* "Repository Administration" (page 188) and *Chapter 6,* "Server Configuration" (page 236) will show you how to create your first repository and make it available over the network. After that's done, *Chapter 2,* "Basic Usage" (page 49) and *Appendix B,* "Subversion for CVS Users" (page 443) are the fastest routes to learning the Subversion client.

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, then *Chapter 1,* "Fundamental Concepts" (page 33) is a vital introduction to the ideas behind version control. *Chapter 2,* "Basic Usage" (page 49) is a guided tour of the Subversion client.

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 Subversion's property support (*Chapter 3,* "Advanced Topics" – page 83), how to use branches and perform merges (*Chapter 4,* "Branching and Merging" – page 143), how to configure runtime options (*Chapter 7,* "Customizing Your Subversion Experience" – page 279), and other things. These chapters aren't critical 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,* "Embedding Subversion" (page 297) is just for you.

The book ends with reference material—*Chapter 9,* "Subversion Complete Reference" (page 318) is a reference guide for all Subversion commands, and the appendixes 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

The following typographic conventions are used in this book:

`Constant width`

Used for literal user input, command output, and command-line options

`Italic`

Used for program and Subversion tool subcommand names, file and directory names, and new terms

`Constant width italic`

Used for replaceable items in code and text

Also, we sprinkled especially helpful or important bits of information throughout the book (in contextually relevant locations), set off visually so they're easy to find. Look for the following icons as you read:

Note

This icon designates a special point of interest.

Tip

This icon designates a helpful tip or recommended best practice.

Warning

This icon designates a warning. Pay close attention to these to avoid running into problems.

Organization of This Book

The chapters that follow and their contents are listed here:

Chapter 1, Fundamental Concepts

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

Chapter 2, Basic Usage

Walks you through a day in the life of a Subversion user. It demonstrates how to use a Subversion client to obtain, modify, and commit data.

Chapter 3, Advanced Topics

Covers more complex features that regular users will eventually come into contact with, such as versioned metadata, file locking, and peg revisions.

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 offers different ways to access your repository: `HTTP`, the `svn` protocol, and local disk access. It also covers the details of authentication, authorization and anonymous access.

Chapter 7, Customizing Your Subversion Experience

Explores the Subversion client configuration files, the handling of internationalized text, and how to make external tools cooperate with Subversion.

Chapter 8, Embedding Subversion

Describes the internals of Subversion, the Subversion filesystem, and the working copy administrative areas from a programmer's point of view. It also demonstrates how to use the public APIs to write a program that uses 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 Quick-Start Guide

For the impatient, a whirlwind explanation of how to install Subversion and start using it immediately. You have been warned.

Appendix B, 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*** versus ***status***, branches, tags, metadata, conflict resolution, and authentication.

Appendix C, 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 D, Copyright

A copy of the Creative Commons Attribution License, under which this book is licensed.

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" – page 460). In fact, the book was written in the public eye, originally as part of the Subversion project itself. This means two things:

- You will always find the latest version of this book in the book's own Subversion repository.
- You can make changes to this book and redistribute it however you wish—it's under a free license. Your only obligation is to maintain proper attribution to the original authors. Of course, we'd much rather you send feedback and patches to the Subversion developer community, instead of distributing your private version of this book.

The online home of this book's development and most of the volunteer-driven translation efforts regarding it is *http://svnbook.red-bean.com*. There you can find links to the latest releases and tagged versions of the book in various formats, as well as instructions for accessing the book's Subversion repository (where its DocBook XML source code lives). Feedback is welcomed—encouraged, even. Please submit all comments, complaints, and patches against the book sources to `<svnbook-dev@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; and Karl Fogel for being such a good friend and a great community leader, in that order.[1]

Thanks to O'Reilly and our various editors: Chuck Toporek, Linda Mui, Tatiana Apandi, Mary Brady, and Mary Treseler. Their patience and support has been tremendous.

Finally, we thank the countless people who contributed to this book with informal reviews, suggestions, and patches. While this is undoubtedly not a complete list, this book would be incomplete and incorrect without their help: Bhuvaneswaran A, David Alber, C. Scott Ananian, David Anderson, Ariel Arjona, Seth Arnold, Jani Averbach, Charles Bailey, Ryan Barrett, Francois Beausoleil, Brian R. Becker, Yves Bergeron, Karl Berry, Jennifer Bevan, Matt Blais, Jim Blandy, Phil Bordelon, Sietse Brouwer, Tom Brown, Zack Brown, Martin Buchholz, Paul Burba, Sean Callan-Hinsvark, Branko Cibej, Archie Cobbs, Jason Cohen, Ryan Cresawn, John R. Daily, Peter Davis, Olivier Davy, Robert P. J. Day, Mo DeJong, Brian Denny, Joe Drew, Markus Dreyer, Nick Duffek, Boris Dusek, Ben Elliston, Justin Erenkrantz, Jens M. Felderhoff, Kyle Ferrio, Shlomi Fish, Julian Foad, Chris Foote, Martin Furter, Vlad Georgescu, Peter Gervai, Dave Gilbert, Eric Gillespie, David Glasser, Marcel Gosselin, Lieven Govaerts, Steve Greenland, Matthew Gregan, Tom Gregory, Maverick Grey, Art Haas, Mark E. Hamilton, Eric Hanchrow, Liam Healy, Malte Helmert, Michael Henderson,

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

Øyvind A. Holm, Greg Hudson, Alexis Huxley, Auke Jilderda, Toby Johnson, Jens B. Jorgensen, Tez Kamihira, David Kimdon, Mark Benedetto King, Robert Kleemann, Erik Kline, Josh Knowles, Andreas J. Koenig, Axel Kollmorgen, Nuutti Kotivuori, Kalin Kozhuharov, Matt Kraai, Regis Kuckaertz, Stefan Kueng, Steve Kunkee, Scott Lamb, Wesley J. Landaker, Benjamin Landsteiner, Vincent Lefevre, Morten Ludvigsen, Dennis Lundberg, Paul Lussier, Bruce A. Mah, Jonathon Mah, Karl Heinz Marbaise, Philip Martin, Feliciano Matias, Neil Mayhew, Patrick Mayweg, Gareth McCaughan, Craig McElroy, Simon McKenna, Christophe Meresse, Jonathan Metillon, Jean-Francois Michaud, Jon Middleton, Robert Moerland, Marcel Molina Jr., Tim Moloney, Alexander Mueller, Tabish Mustufa, Christopher Ness, Roman Neuhauser, Mats Nilsson, Greg Noel, Joe Orton, Eric Paire, Dimitri Papadopoulos-Orfanos, Jerry Peek, Chris Pepper, Amy Lyn Pilato, Kevin Pilch-Bisson, Hans Polak, Dmitriy Popkov, Michael Price, Mark Proctor, Steffen Prohaska, Daniel Rall, Srinivasa Ramanujan, Jack Repenning, Tobias Ringstrom, Jason Robbins, Garrett Rooney, Joel Rosdahl, Christian Sauer, Ryan Schmidt, Jochem Schulenklopper, Jens Seidel, Daniel Shahaf, Larry Shatzer, Danil Shopyrin, Erik Sjoelund, Joey Smith, W. Snyder, Stefan Sperling, Robert Spier, M. S. Sriram, Russell Steicke, David Steinbrunner, Sander Striker, David Summers, Johan Sundstroem, Ed Swierk, John Szakmeister, Arfrever Frehtes Taifersar Arahesis, Robert Tasarz, Michael W. Thelen, Mason Thomas, Erik van der Kolk, Joshua Varner, Eric Wadsworth, Chris Wagner, Colin Watson, Alex Waugh, Chad Whitacre, Andy Whitcroft, Josef Wolf, Luke Worth, Hyrum Wright, Blair Zajac, Florian Zumbiehl, and the entire Subversion community.

From Ben Collins-Sussman

Thanks to my wife Frances, who, for many months, got to hear “But honey, I'm still working on the book,” rather than the usual “But honey, I'm still doing email.” I don't know where she gets all that patience! She's my perfect counterbalance.

Thanks to my extended family and friends for their sincere encouragement, despite having no actual interest in the subject. (You know, the ones who say, “Ooh, you wrote a book?” and then when you tell them it's a computer book, sort of glaze over.)

Thanks to all my close friends, who make me a rich, rich man. Don't look at me that way—you know who you are.

Thanks to my parents for the perfect low-level formatting and being unbelievable role models. Thanks to my kids for the opportunity to pass that on.

From Brian W. Fitzpatrick

Huge thanks to my wife Marie for being incredibly understanding, supportive, and most of all, patient. Thank you to my brother Eric who first introduced me to Unix programming way back when. Thanks to my Mom and Grandmother for all their support, not to mention

enduring a Christmas holiday where I came home and promptly buried my head in my laptop to work on the book.

To Mike and Ben: it was a pleasure working with you on the book. Heck, it's a pleasure working with you at work!

To everyone in the Subversion community and the Apache Software Foundation, thanks for having me. Not a day goes by where I don't learn something from at least one of you.

Lastly, thanks to my grandfather, who always told me that "freedom equals responsibility." I couldn't agree more.

From C. Michael Pilato

Special thanks to Amy, my best friend and wife of more than ten incredible years, for her love and patient support, for putting up with the late nights, and for graciously enduring the version control processes I've imposed on her. Don't worry, sweetheart—you'll be a TortoiseSVN wizard in no time!

Gavin, you're able to read half of the words in this book yourself now; sadly, it's the other half that provide the key concepts. And sorry, Aidan—I couldn't find a way to work Disney/Pixar characters into the text. But Daddy loves you both, and can't wait to teach you about programming.

Mom and Dad, thanks for your constant support and enthusiasm. Mom- and Dad-in-law, thanks for all of the same *plus* your fabulous daughter.

Hats off to Shep Kendall, through whom the world of computers was first opened to me; Ben Collins-Sussman, my tour guide through the open source world; Karl Fogel, you *are* my `.emacs`; Greg Stein, for oozing practical programming know-how; and Brian Fitzpatrick, for sharing this writing experience with me. To the many folks from whom I am constantly picking up new knowledge—keep dropping it!

Finally, to the One who perfectly demonstrates creative excellence—thank You.

What Is Subversion?

Subversion is a free/open source version control system. That is, Subversion manages files and directories, and the changes made to them, over time. This allows you to recover older versions of your data or examine the history of how your data changed. In this regard, many people think of a version control system as a sort of "time machine."

Subversion can operate across networks, which allows it to be used by people on different computers. At some level, the ability for various people to modify and manage the same set of data from their respective locations fosters collaboration. Progress can occur more quickly without a single conduit through which all modifications must occur. And because the work

is versioned, you need not fear that quality is the trade-off for losing that conduit—if some incorrect change is made to the data, just undo that change.

Some version control systems are also software configuration management (SCM) systems. These systems are specifically tailored to manage trees of source code and have many features that are specific to software development—such as natively understanding programming languages, or supplying tools for building software. Subversion, however, is not one of these systems. It is a general system that can be used to manage *any* collection of files. For you, those files might be source code—for others, anything from grocery shopping lists to digital video mixdowns and beyond.

Is Subversion the Right Tool?

If you're a user or system administrator pondering the use of Subversion, the first question you should ask yourself is: "Is this the right tool for the job?" Subversion is a fantastic hammer, but be careful not to view every problem as a nail.

If you need to archive old versions of files and directories, possibly resurrect them, or examine logs of how they've changed over time, then Subversion is exactly the right tool for you. If you need to collaborate with people on documents (usually over a network) and keep track of who made which changes, then Subversion is also appropriate. This is why Subversion is so often used in software development environments—working on a development team is an inherently social activity, and Subversion makes it easy to collaborate with other programmers. Of course, there's a cost to using Subversion as well: administrative overhead. You'll need to manage a data repository to store the information and all its history, and be diligent about backing it up. When working with the data on a daily basis, you won't be able to copy, move, rename, or delete files the way you usually do. Instead, you'll have to do all of those things through Subversion.

Assuming you're fine with the extra workflow, you should still make sure you're not using Subversion to solve a problem that other tools solve better. For example, because Subversion replicates data to all the collaborators involved, a common misuse is to treat it as a generic distribution system. People will sometimes use Subversion to distribute huge collections of photos, digital music, or software packages. The problem is that this sort of data usually isn't changing at all. The collection itself grows over time, but the individual files within the collection aren't being changed. In this case, using Subversion is "overkill."[2] There are simpler tools that efficiently replicate data *without* the overhead of tracking changes, such as ***rsync*** or ***unison***.

[2] Or as a friend puts it, "swatting a fly with a Buick."

Subversion's History

In early 2000, CollabNet, Inc. (*http://www.collab.net*) began seeking developers to write a replacement for CVS. CollabNet offers a collaboration software suite called CollabNet Enterprise Edition (CEE), of which one component is version control. Although CEE used CVS as its initial version control system, CVS's limitations were obvious from the beginning, and CollabNet knew it would eventually have to find something better. Unfortunately, CVS had become the de facto standard in the open source world largely because there *wasn't* anything better, at least not under a free license. So CollabNet determined to write a new version control system from scratch, retaining the basic ideas of CVS, but without the bugs and misfeatures.

In February 2000, they contacted Karl Fogel, the author of *Open Source Development with CVS* (Coriolis, 1999), and asked if he'd like to work on this new project. Coincidentally, at the time Karl was already discussing a design for a new version control system with his friend Jim Blandy. In 1995, the two had started Cyclic Software, a company providing CVS support contracts, and although they later sold the business, they still used CVS every day at their jobs. Their frustration with CVS had led Jim to think carefully about better ways to manage versioned data, and he'd already come up with not only the name "Subversion," but also the basic design of the Subversion data store. When CollabNet called, Karl immediately agreed to work on the project, and Jim got his employer, Red Hat Software, to essentially donate him to the project for an indefinite period of time. CollabNet hired Karl and Ben Collins-Sussman, and detailed design work began in May 2000. With the help of some well-placed prods from Brian Behlendorf and Jason Robbins of CollabNet, and from 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 encountered 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 14 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 Architecture

Figure 1, "Subversion's architecture" illustrates a "mile-high" view of Subversion's design.

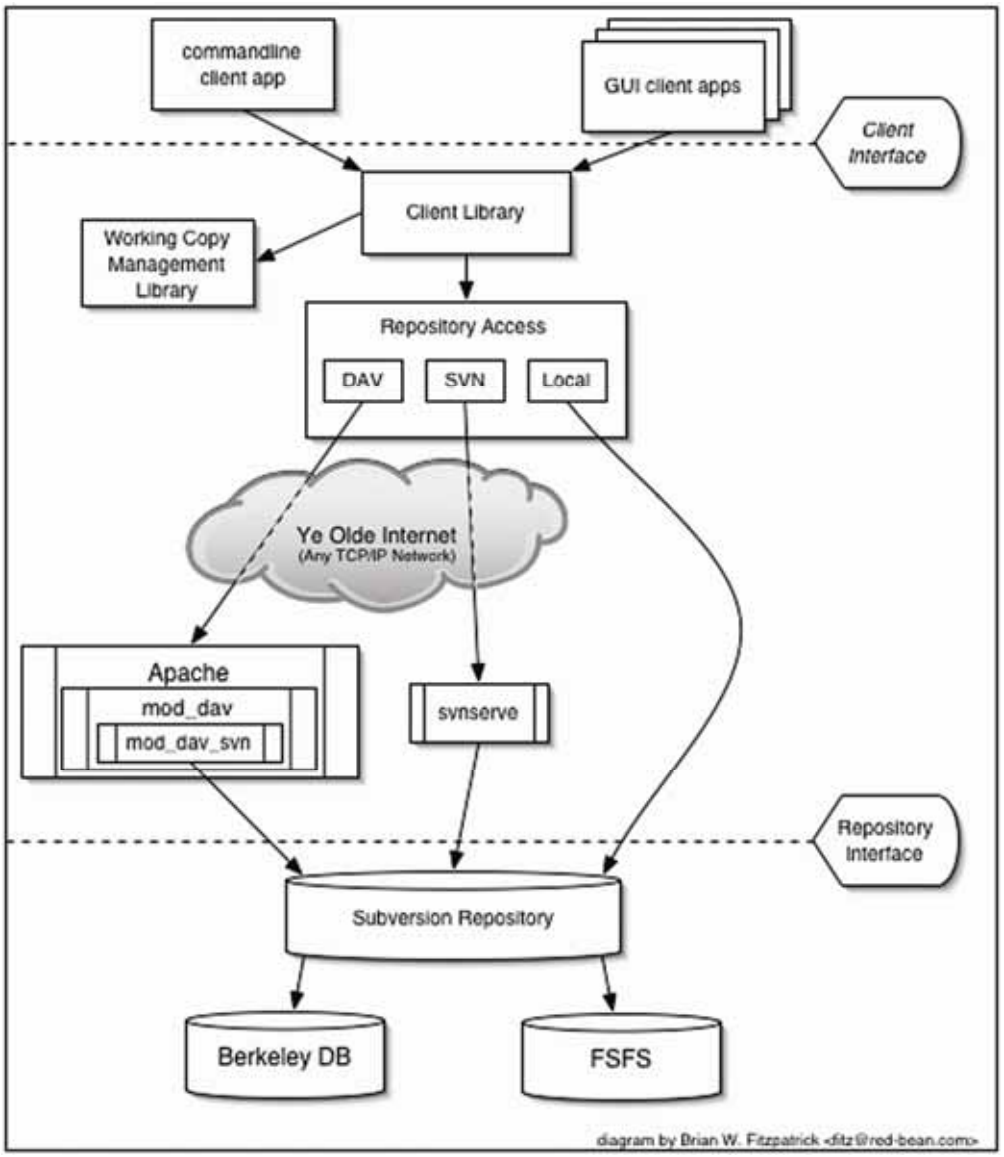

Figure 1. Subversion's architecture

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.

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—*plenty* more pages in this book are 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 directly inspecting a Subversion repository

svnadmin

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

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.

svndumpfilter

: A program for filtering Subversion repository dump streams

svnsync

: A program for incrementally mirroring one repository to another over a network

What's New in Subversion

The first edition of this book was released in 2004, shortly after Subversion had reached 1.0. Over the following four years Subversion released five major new versions, fixing bugs and

adding major new features. While we've managed to keep the online version of this book up to date, we're thrilled that the second edition from O'Reilly now covers Subversion up through release 1.5, a major milestone for the project. Here's a quick summary of major new changes since Subversion 1.0. Note that this is not a complete list; for full details, please visit Subversion's web site at *http://subversion.tigris.org*.

Subversion 1.1 (September 2004)

> Release 1.1 introduced FSFS, a flat-file repository storage option for the repository. While the Berkeley DB backend is still widely used and supported, FSFS has since become the default choice for newly created repositories due to its low barrier to entry and minimal maintenance requirements. Also in this release came the ability to put symbolic links under version control, auto-escaping of URLs, and a localized user interface.

Subversion 1.2 (May 2005)

> Release 1.2 introduced the ability to create server-side locks on files, thus serializing commit access to certain resources. While Subversion is still a fundamentally concurrent version control system, certain types of binary files (e.g. art assets) cannot be merged together. The locking feature fulfills the need to version and protect such resources. With locking also came a complete WebDAV auto-versioning implementation, allowing Subversion repositories to be mounted as network folders. Finally, Subversion 1.2 began using a new, faster binary-differencing algorithm to compress and retrieve old versions of files.

Subversion 1.3 (December 2005)

> Release 1.3 brought path-based authorization controls to the ***svnserve*** server, matching a feature formerly found only in the Apache server. The Apache server, however, gained some new logging features of its own, and Subversion's API bindings to other languages also made great leaps forward.

Subversion 1.4 (September 2006)

> Release 1.4 introduced a whole new tool—***svnsync***—for doing one-way repository replication over a network. Major parts of the working copy metadata were revamped to no longer use XML (resulting in client-side speed gains), while the Berkeley DB repository backend gained the ability to automatically recover itself after a server crash.

Subversion 1.5 (June 2008)

> Release 1.5 took much longer to finish than prior releases, but the headliner feature was gigantic: semi-automated tracking of branching and merging. This was a huge boon for users, and pushed Subversion far beyond the abilities of CVS and into the

ranks of commercial competitors such as Perforce and ClearCase. Subversion 1.5 also introduced a bevy of other user-focused features, such as interactive resolution of file conflicts, partial checkouts, client-side management of changelists, powerful new syntax for externals definitions, and SASL authentication support for the ***svnserve*** server.

Subversion 1.6 (March 2009)

Major new features in the latest release include the ability to detect tree conflicts, improved management of credentials, and reduced repository space requirements.

Chapter 1.
Fundamental 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.

1.1. 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 1.1,* "A typical client/server system" illustrates this.

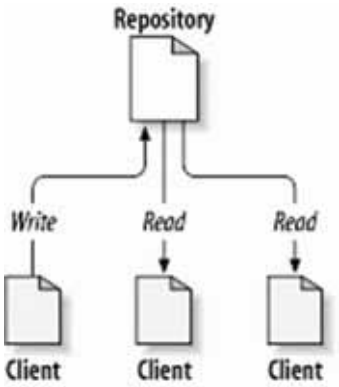

Figure 1.1. A typical client/server system

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 such as "What did this directory contain last Wednesday?" and "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 track changes to data over time.

1.2. 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. It's important to understand these different strategies, for a couple of reasons. First, it will help you compare and contrast existing version control systems, in case you encounter other systems similar to Subversion. Beyond that, it will also help you make more effective use of Subversion, since Subversion itself supports a couple of different ways of working.

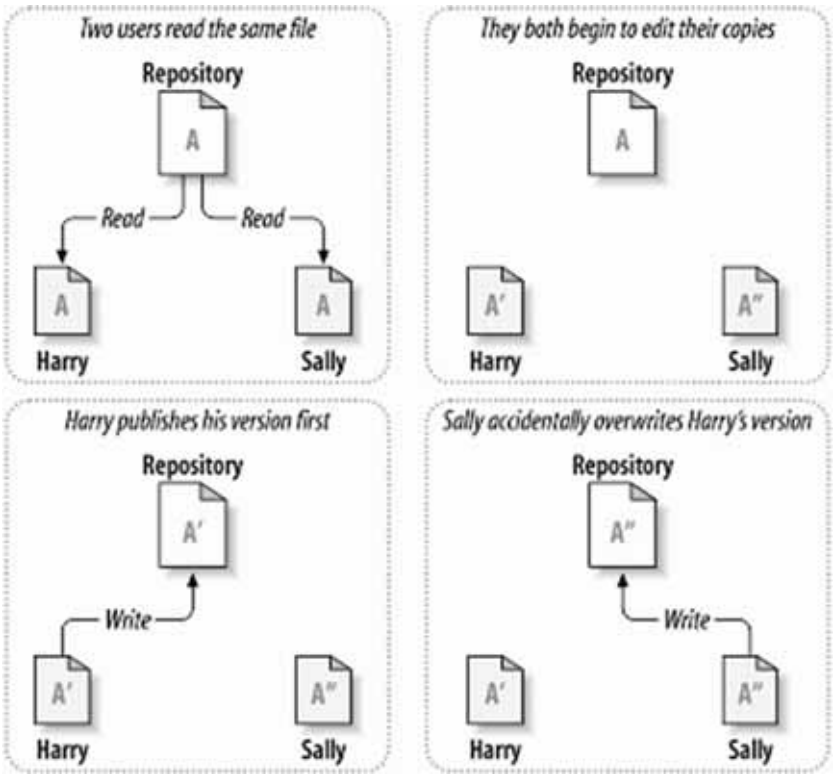

Figure 1.2. The problem to avoid

1.2.1. 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 1.2,* "The problem to avoid". Suppose we have two coworkers, 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, 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!

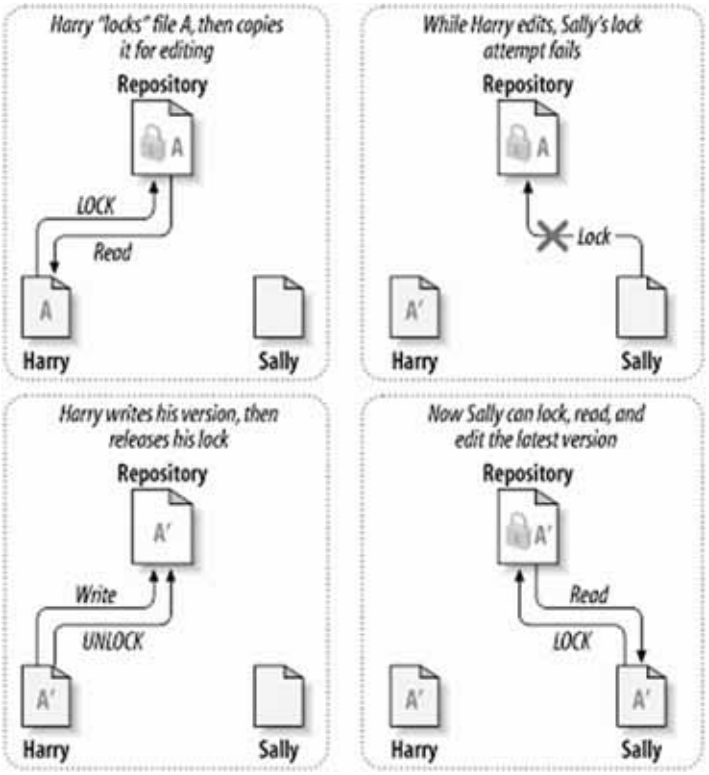

Figure 1.3. The lock-modify-unlock solution

1.2.2. 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, 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 1.3,* "The lock-modify-unlock solution" demonstrates this simple solution.

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

- *Locking may cause administrative problems.* Sometimes Harry will lock a file and then forget about it. Meanwhile, because Sally is still waiting to edit the file, her hands are tied. And then Harry goes on vacation. Now Sally has to get an administrator to release Harry's lock. The situation ends up causing a lot of unnecessary delay and wasted time.
- *Locking may cause unnecessary serialization.* What if Harry is editing the beginning of a text file, and Sally simply wants to edit the end of the same file? These changes don't overlap at all. They could easily edit the file simultaneously, and no great harm would come, assuming the changes were properly merged together. There's no need for them to take turns in this situation.
- *Locking may create a false sense of security.* Suppose Harry locks and edits file A, while Sally simultaneously locks and edits file B. But what if A and B depend on one another, and the changes made to each are semantically incompatible? Suddenly A and B don't work together anymore. 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 they need not bother discussing their incompatible changes early on. Locking often becomes a substitute for real communication.

1.2.3. The Copy-Modify-Merge Solution

Subversion, CVS, and many 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 simultaneously and independently, 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, 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; once he has both sets of changes integrated, he saves his working copy back to the repository. *Figure 1.4,* "The copy-modify-merge solution" and *Figure 1.5,* "The copy-modify-merge solution (continued)" show this process.

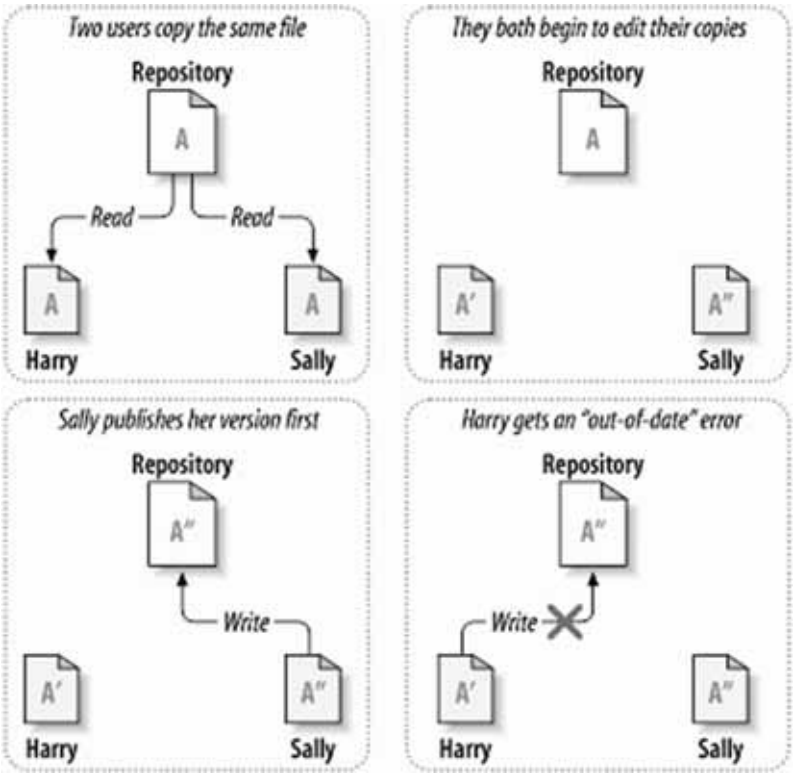

Figure 1.4. The copy-modify-merge solution

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.

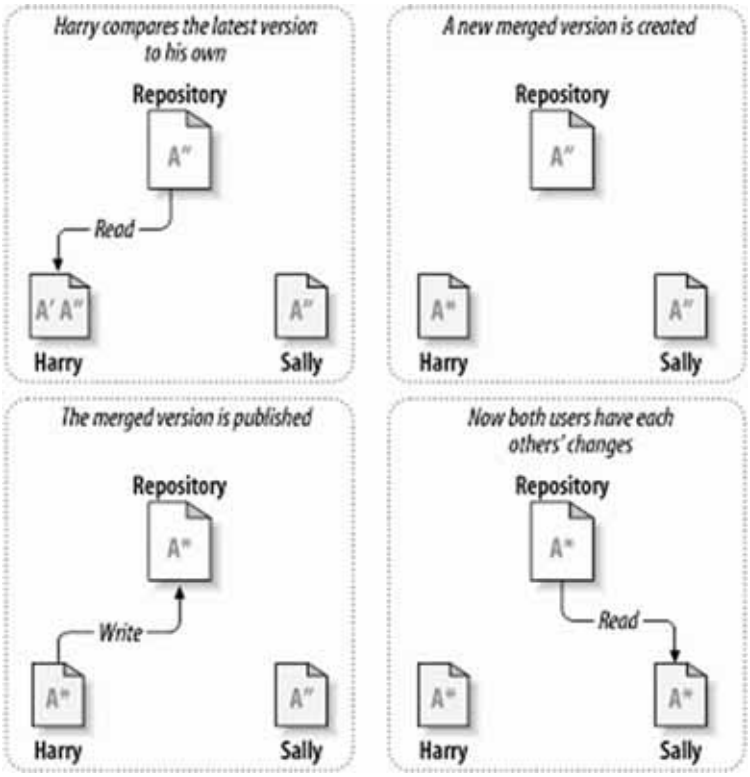

Figure 1.5. The copy-modify-merge solution (continued)

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 usually 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 sense of security 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, sometimes locking is appropriate.

When Locking Is Necessary

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 for users to take strict turns when changing the file. Without serialized access, somebody ends up wasting time on changes that are ultimately discarded.

While Subversion is primarily a copy-modify-merge system, it still recognizes the need to lock an occasional file, and thus provides mechanisms for this. We discuss this feature in the section called "Locking" (page 112).

1.3. 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.

1.3.1. Subversion Repository URLs

Throughout this book, Subversion uses URLs to identify versioned files and directories in Subversion repositories. For the most part, these URLs use the standard syntax, allowing for server names and port numbers to be specified as part of the URL:

```
$ svn checkout http://svn.example.com:9834/repos
…
```

But there are some nuances in Subversion's handling of URLs that are notable. For example, URLs containing the `file://` access method (used for local repositories) must, in accordance with convention, have either a server name of `localhost` or no server name at all:

```
$ svn checkout file:///var/svn/repos
…
$ svn checkout file://localhost/var/svn/repos
…
```

Also, users of the `file://` scheme on Windows platforms will need to use an unofficially "standard" syntax for accessing repositories that are on the same machine, but on a different drive than the client's current working drive. Either of the two following URL path syntaxes will work, where `X` is the drive on which the repository resides:

```
C:\> svn checkout file:///X:/var/svn/repos
…
C:\> svn checkout "file:///X|/var/svn/repos"
…
```

In the second syntax, you need to quote the URL so that the vertical bar character is not interpreted as a pipe. Also, note that a URL uses forward slashes even though the native (non-URL) form of a path on Windows uses backslashes.

Note

You cannot use Subversion's `file://` URLs 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" - page 299), and your browser will not understand how to interact with that filesystem.

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 as in the following:

```
$ svn checkout "http://host/path with space/project/españa"
```

then Subversion will escape the unsafe characters and behave as though 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 quotation marks so that your shell treats the whole thing as a single argument to the ***svn*** program.

In Subversion 1.6, a new caret (^) notation was introduced as a shorthand for "the URL of the repository's root directory". For example:

```
$ svn list ^/tags/bigsandwich/
```

In this example, we're specifying a URL for the `/tags/bigsandwich` directory in the root of the repository. Note that this URL syntax *only* works when your current working directory is a working copy—the commandline client knows the repository's root URL by looking at the working copy's metadata.

Repository URLs

You can access Subversion repositories through many different methods—on local disk or through various network protocols, depending on how your administrator has set things up for you. A repository location, however, is always a URL. *Table 1.1,* "Repository access URLs" describes how different URL schemes map to the available access methods.

Schema	Access method
`file:///`	Direct repository access (on local disk)
`http://`	Access via WebDAV protocol to Subversion-aware Apache server

Repository URLs

Schema	Access method
`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.

Table 1.1. Repository access URLs

For more information on how Subversion parses URLs, see the section called "Subversion Repository URLs" (page 39). For more information on the different types of network servers available for Subversion, see *Chapter 6,* "Server Configuration" (page 236).

1.3.2. 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 copy (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's *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 1.6,* "The repository's filesystem".

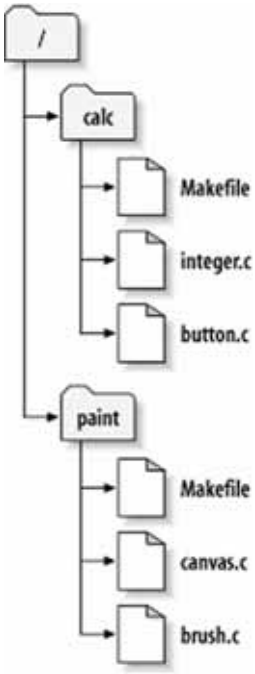

Figure 1.6. The repository's filesystem

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  button.c integer.c .svn/
```

The list of letter `A`s in the left margin 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.

Suppose you make changes to `button.c`. Since the `.svn` directory remembers the file's original modification date and 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 ***svn commit*** command:

```
$ svn commit button.c -m "Fixed a typo in button.c."
Sending        button.c
Transmitting file data .
Committed revision 57.
```

Now your changes to *button.c* have been committed to the repository, with a note describing your change (namely, that you fixed a typo). If another user checks out a working copy of */calc*, she 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 modifies working copies only at the user's request.

To bring her project up to date, Sally can ask Subversion to *update* her working copy, by using the ***svn 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
Makefile button.c integer.c .svn/

$ 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 as well as further information in the repository, to decide which files need to be brought up to date.

1.3.3. Revisions

An ***svn commit*** operation publishes 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 a complete set of changes as an atomic transaction.

By atomic transaction, we mean simply this: either all of the changes happen in the repository, or none of them happens. 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 0 and consists of nothing but an empty root directory.

Figure 1.7, "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.

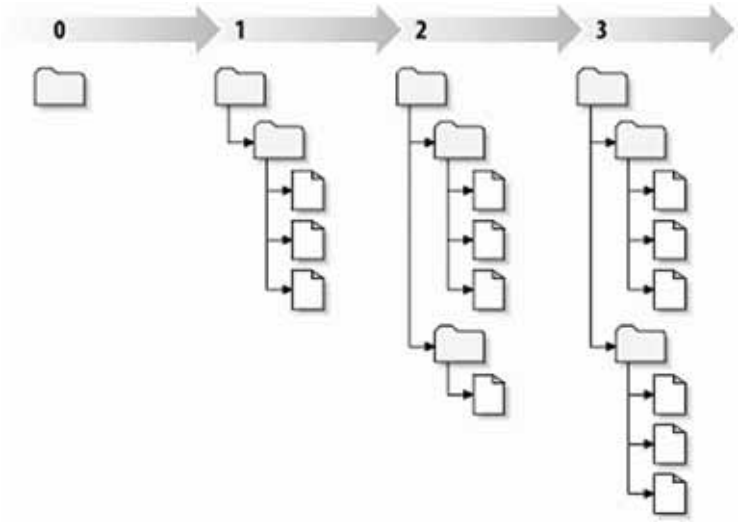

Figure 1.7. The repository

Global Revision Numbers
Unlike most version control systems, Subversion's revision numbers apply to *entire trees*, not individual files. Each revision number selects an entire tree, a particular state of the repository after some committed change. Another way to think about it is that revision N represents the state of the repository filesystem after the Nth commit. When Subversion users talk about "revision 5 of `foo.c`," they really mean "`foo.c` as it appears in revision 5." Notice that in general, revisions N and M of a file do *not* necessarily differ! Many other version control systems use per-file revision numbers, so this concept may seem unusual at first. (Former CVS users might want to see *Appendix B,* "Subversion for CVS Users" (page 443) for more details.)

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

```
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, 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.

1.3.4. 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*)
- 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 you last updated. 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 in order to make it current with the latest 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 "See an overview of your changes" (page 58).

1.3.5. 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 why the feature exists and how to make use of it.

1.3.5.1. Updates and commits are separate

One of the fundamental rules of Subversion is that a "push" action does not cause a "pull," nor vice versa. 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, ***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 as well as be tolerant of the mixture. 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, 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.

1.3.5.2. 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*** command with the `--verbose` (`-v`) option (see the section called "See an overview of your changes" (page 58) 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 "Generating a List of Historical Changes" – page 71). When the user invokes this command on a working copy object, he expects 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), the history of the *older* version of the object is shown.

1.3.5.3. Mixed revisions are useful

If your project is sufficiently complex, you'll discover that it's sometimes nice to forcibly *backdate* (or update to a revision older than the one you already have) portions of your working copy to an earlier revision; you'll learn how to do that in *Chapter 2,* "Basic Usage" (page 49). Perhaps you'd like to test an earlier version of a submodule 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 that allows you to move any portion of your working copy forward and backward in history.

1.3.5.4. 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 that 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 3,* "Advanced Topics" (page 83). 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.

1.4. Summary

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

- We introduced the notions of the central repository, the client working copy, and the array of repository revision trees.
- We saw 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 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 move into the next chapter, which is a detailed tour of Subversion's commands and features.

Appendix D. Copyright

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.

1. Definitions

   a. "Collective Work" means a work, such as a periodical issue, anthology or encyclopedia, in which the Work in its entirety in unmodified form, along with a number of other contributions, constituting separate and independent works in themselves, are assembled into a collective whole. A work that constitutes a Collective Work will not be considered a Derivative Work (as defined below) for the purposes of this License.

   b. "Derivative Work" means a work based upon the Work or upon the Work and other pre-existing works, such as a translation, musical arrangement, dramatization, fictionalization, motion picture version, sound recording, art reproduction, abridgment, condensation, or any other form in which the Work may be recast, transformed, or adapted, except that a work that constitutes a Collective Work will not be considered a Derivative Work for the purpose of this License. For the avoidance of doubt, where the Work is a musical composition or sound recording, the synchronization of the Work in timed-relation with a moving image ("synching") will be considered a Derivative Work for the purpose of this License.

   c. "Licensor" means the individual or entity that offers the Work under the terms of this License.

   d. "Original Author" means the individual or entity who created the Work.

   e. "Work" means the copyrightable work of authorship offered under the terms of this License.

   f. "You" means an individual or entity exercising rights under this License who has not previously violated the terms of this License with respect to the Work, or who has received express permission from the Licensor to exercise rights under this License despite a previous violation.

2. Fair Use Rights. Nothing in this license is intended to reduce, limit, or restrict any rights arising from fair use, first sale or other limitations on the exclusive rights of the copyright owner under copyright law or other applicable laws.

```
3. License Grant. Subject to the terms and conditions of this License,
   Licensor hereby grants You a worldwide, royalty-free,
   non-exclusive, perpetual (for the duration of the applicable
   copyright) license to exercise the rights in the Work as stated below:

   a. to reproduce the Work, to incorporate the Work into one or more
      Collective Works, and to reproduce the Work as incorporated in
      the Collective Works;

   b. to create and reproduce Derivative Works;

   c. to distribute copies or phonorecords of, display publicly,
      perform publicly, and perform publicly by means of a digital
      audio transmission the Work including as incorporated in
      Collective Works;

   d. to distribute copies or phonorecords of, display publicly,
      perform publicly, and perform publicly by means of a digital
      audio transmission Derivative Works.

   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 Licensor or the Original Author, as requested. If You create a Derivative Work, upon notice from any Licensor You must, to the extent practicable, remove from the Derivative Work any reference to such Licensor or the Original Author, as requested.

b. If you distribute, publicly display, publicly perform, or publicly digitally perform the Work or any Derivative Works or Collective Works, You must keep intact all copyright notices for the Work and give the Original Author credit reasonable to the medium or means You are utilizing by conveying the name (or pseudonym if applicable) of the Original Author if supplied; the title of the Work if supplied; to the extent reasonably practicable, the Uniform Resource Identifier, if any, that Licensor specifies to be associated with the Work, unless such URI does not refer to the copyright notice or licensing information for the Work; and in the case of a Derivative Work, a credit identifying the use of the Work in the Derivative Work (e.g., "French translation of the Work by Original Author," or "Screenplay based on original Work by Original Author"). Such credit may be implemented in any reasonable manner; provided, however, that in the case of a Derivative Work or Collective Work, at a minimum such credit will appear where any other comparable authorship credit appears and in a manner at least as prominent as such other comparable authorship credit.

5. Representations, Warranties and Disclaimer

UNLESS OTHERWISE MUTUALLY AGREED TO BY THE PARTIES IN WRITING, LICENSOR OFFERS THE WORK AS-IS AND MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND CONCERNING THE WORK, EXPRESS, IMPLIED, STATUTORY OR OTHERWISE, INCLUDING, WITHOUT LIMITATION, WARRANTIES OF TITLE, MERCHANTIBILITY, FITNESS FOR A PARTICULAR PURPOSE, NONINFRINGEMENT, OR THE ABSENCE OF LATENT OR OTHER DEFECTS, ACCURACY, OR THE PRESENCE OF ABSENCE OF ERRORS, WHETHER OR NOT DISCOVERABLE. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO SUCH EXCLUSION MAY NOT APPLY TO YOU.

```
6. Limitation on Liability. EXCEPT TO THE EXTENT REQUIRED BY
   APPLICABLE LAW, IN NO EVENT WILL LICENSOR BE LIABLE TO YOU ON ANY
   LEGAL THEORY FOR ANY SPECIAL, INCIDENTAL, CONSEQUENTIAL, PUNITIVE
   OR EXEMPLARY DAMAGES ARISING OUT OF THIS LICENSE OR THE USE OF THE
   WORK, EVEN IF LICENSOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH
   DAMAGES.

7. Termination

   a. This License and the rights granted hereunder will terminate
      automatically upon any breach by You of the terms of this
      License. Individuals or entities who have received Derivative
      Works or Collective Works from You under this License, however,
      will not have their licenses terminated provided such
      individuals or entities remain in full compliance with those
      licenses. Sections 1, 2, 5, 6, 7, and 8 will survive any
      termination of this License.

   b. Subject to the above terms and conditions, the license granted
      here is perpetual (for the duration of the applicable copyright
      in the Work). Notwithstanding the above, Licensor reserves the
      right to release the Work under different license terms or to
      stop distributing the Work at any time; provided, however that
      any such election will not serve to withdraw this License (or
      any other license that has been, or is required to be, granted
      under the terms of this License), and this License will continue
      in full force and effect unless terminated as stated above.

8. Miscellaneous

   a. Each time You distribute or publicly digitally perform the Work
      or a Collective Work, the Licensor offers to the recipient a
      license to the Work on the same terms and conditions as the
      license granted to You under this License.

   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
      license granted to You under this License.

   c. If any provision of this License is invalid or unenforceable
      under applicable law, it shall not affect the validity or
      enforceability of the remainder of the terms of this License,
      and without further action by the parties to this agreement,
      such provision shall be reformed to the minimum extent necessary
      to make such provision valid and enforceable.

   d. No term or provision of this License shall be deemed waived and
      no breach consented to unless such waiver or consent shall be in
      writing and signed by the party to be charged with such waiver
      or consent.
```

```
   e. This License constitutes the entire agreement between the
      parties with respect to the Work licensed here. There are no
      understandings, agreements or representations with respect to
      the Work not specified here. Licensor shall not be bound by any
      additional provisions that may appear in any communication from
      You. This License may not be modified without the mutual written
      agreement of the Licensor and You.

Creative Commons is not a party to this License, and makes no warranty whatsoever
in connection with the Work. Creative Commons will not be liable to You or any
party on any legal theory for any damages whatsoever, including without limitation
any general, special, incidental or consequential damages arising in connection to
this license. Notwithstanding the foregoing two (2) sentences, if Creative Commons
has expressly identified itself as the Licensor hereunder, it shall have all rights
and obligations of Licensor.

Except for the limited purpose of indicating to the public that the Work is
licensed under the CCPL, neither party will use the trademark "Creative Commons" or
any related trademark or logo of Creative Commons without the prior written consent
of Creative Commons. Any permitted use will be in compliance with Creative Commons'
then-current trademark usage guidelines, as may be published on its website or
otherwise made available upon request from time to time.

Creative Commons may be contacted at http://creativecommons.org/.

======================================================================
```

Index