

The guide to source control

SAMPLE CHAPTER

Subversion IN ACTION

Jeffrey Machols

 MANNING



contents

- Chapter 1 ■ Introduction
- Chapter 2 ■ Getting started
- Chapter 3 ■ Managing your working copy
- Chapter 4 ■ Getting change information
- Chapter 5 ■ Branches and tags
- Chapter 6 ■ Properties
- Chapter 7 ■ Repository administration
- Chapter 8 ■ Advanced administration and configuration
- Chapter 9 ■ Subversion utility clients
- Chapter 10 ■ Third-party tools
- Chapter 11 ■ Subversion in development lifecycle

- Appendix A ■ SSL certificates
- Appendix B ■ Building Subversion

Introduction

In this chapter

- Introduction to version control
- Overview of Subversion
- New revision numbering paradigm

I doubt this was planned, but the dictionary definition of “subversion” happens to provide a fitting description of Subversion’s social role: by *subverting* the commanding position held for years by Concurrent Versions System (CVS), it is quickly becoming the de facto standard for open source version control systems. Many open source development communities are moving away from CVS and adapting Subversion, and this trend will only accelerate. If you want to be in on this trend—and take advantage of the best versioning system available for your group-development infrastructure—this book is for you.

While other areas of software development such as IDE’s progressed, version control remained relatively stagnant, especially in the open source world. But as the practice of community software development caught on, it became apparent that the glue that allows people to work together in different places and at different times is their version control system. It also became clear that the old standard, CVS, was becoming inadequate for supporting advanced open source development practices. From this need, Subversion was born as a product built by developers to make version control a seamless part of the development process instead of an impediment.

If you find yourself involved in a live discussion that touches on Subversion, you might like to know that those in the know do *not* pronounce it as a single word, with the accent on “ver.” Instead, they say it as if it were two words, *Sub version*. Pronounce it as they do and your standing automatically improves.

Now, having finished with that important matter, let’s turn to substance. Throughout this book, we will explore the features of Subversion that are making it the success it is and causing development communities and businesses to embrace it so quickly. Not only will we look at how the system works and what the commands do, but we will also tie it all in with your software development process.

1.1 Understanding version control

First, it will help to understand the basics of a version control system in a generic sense. Think back to elementary school and remember the bespectacled figure of your librarian. She may have been stern but she knew where every book in the library was, who had it checked out, and when a new edition was coming in. A version control system plays a similar role, but without the thick glasses and the nasty shushing, of course. Rather than tracking books in a library, a versioning system manages files in a directory. It controls the files coming in and out and keeps track of who made the change. It also provides the useful capability to get back an old revision of a file.

1.1.1 Why use version control

Have you ever been in a situation where something in a file changed and you asked, “What did that file look like before—when it worked? When did this file change? Who changed it?” I often find myself looking at an edit I made in the past and wondering why I made it. A version control system gives you the answer to these questions. It also gives you the tools to manage your code better by providing features such as the following:

- Helps you manage code by tagging specific versions as a release of the software
- Allows you to add automation to manual tasks through the use of hook scripts, which run when triggered from an action
- Allows for multiple users to work on the same file without losing any changes
- Provides the ability to start another development path from the same code base and then merge the two paths back together
- Efficiently stores multiple versions of a file by keeping only the changes to each version instead of an entire copy

Let’s take a look at how some of these features are implemented by exploring some of the technical aspects of a version control system.

1.1.2 The repository

The core of any version control system is the *repository*, which is basically a souped-up filesystem. It tracks the changes to files and allows multiple users concurrent access. Each time a modified file is saved to the repository, a new *revision* of that file is created. These revisions contain not only the contents in the file that changed but also external information, such as who changed it and when it was changed. In a standard filesystem, you access and edit files directly; not so in a repository. Instead, the system has a client interface that does the reading and writing to the repository for you. This client talks to the repository and allows for the transparent implementation of all the features we just described.

1.1.3 Checkout strategies

We know repositories do not allow direct access to the files and directories, so you need a way to get at your code. To accomplish this, you perform a *checkout*, which is a request for a copy of the file through the client. You can check out one file, a directory, or the entire repository. This is similar to checking out a book from a library. You go to a central location and get a copy of a book, or in the case of a

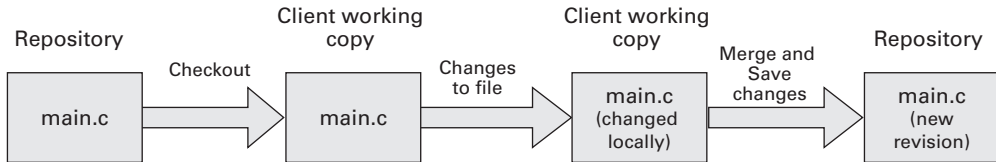


Figure 1.1 Workflow in the copy-modify-merge model

repository, a file. If a new edition or revision comes out, you go back and check out the new one.

Copy-modify-merge. Many version control systems, including Subversion, use a checkout strategy called *copy-modify-merge*. In this model, when you perform a checkout, you get a working copy of the file in a local directory on your machine. This is a snapshot of the repository where you will make your changes independently of other developers. When you have finished with your changes, you copy the file, `main.c` in this example, back into the repository. Figure 1.1 illustrates this workflow.

Then, when another developer checks out the file `main.c`, it will include your changes, and that person will follow the same process. If you are making changes at the same time as someone else, the flow will look something like figure 1.2.

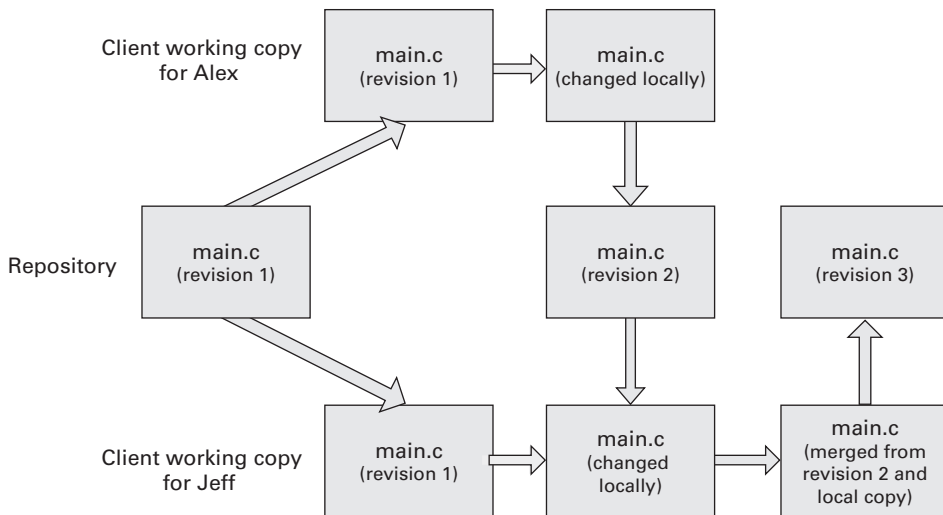


Figure 1.2 Workflow in the copy-modify-merge model with multiple users

This merge will happen automatically if the two changes do not conflict, which would happen if the users made changes to the same line in the file. It is important to understand that these conflicts are strictly based on whether or not the system can merge the data within the file. There are no logical checks that take place to ensure that you did not muck up the source code, so you are responsible for the logical validation of the change.

1.1.4 Commit and update

Version control systems that use the copy-modify-merge model create a working copy of the repository for you to make your changes. You also need a way to save your edits back to the repository. The process of applying your changes into the repository is called *committing* (some systems call this *check in*). When you perform a commit from your working copy, the system will find the files that have changed and compare them against the latest version in the repository. If there are no conflicts, a new revision of the files will be created and saved in the repository. Once you commit, the repository will be in sync with the changes in your working copy, but this is only half the battle.

If you develop in a vacuum, the checkout and commit give you everything needed to keep the repository and working copy in sync, but this does not happen in the real world. While you were editing your working copy, Alex committed his changes to the repository. So not only do you need a way to get your changes into the repository, you need a way to get any changes made to the repository back into your working copy; this is done with an update. An *update* is a specialized checkout that gives you only the changes in the repository since your checkout or last update.

You know that a commit saves your changes back to the repository, but remember that there is more to a version control system than just file contents. When the commit occurs, a record of the change is also saved. This record is called a *change log* and is one of the components that gives the answer to those who, when, and why questions we have been trying to answer.

1.1.5 What are change logs?

One of the reasons for using a version control system is to get more information about a change than just the content difference in a file between revisions. This additional information is stored in the change log and is attached to each revision of a file. Let's take a look at a typical change for the file `main.c` in figure 1.3.

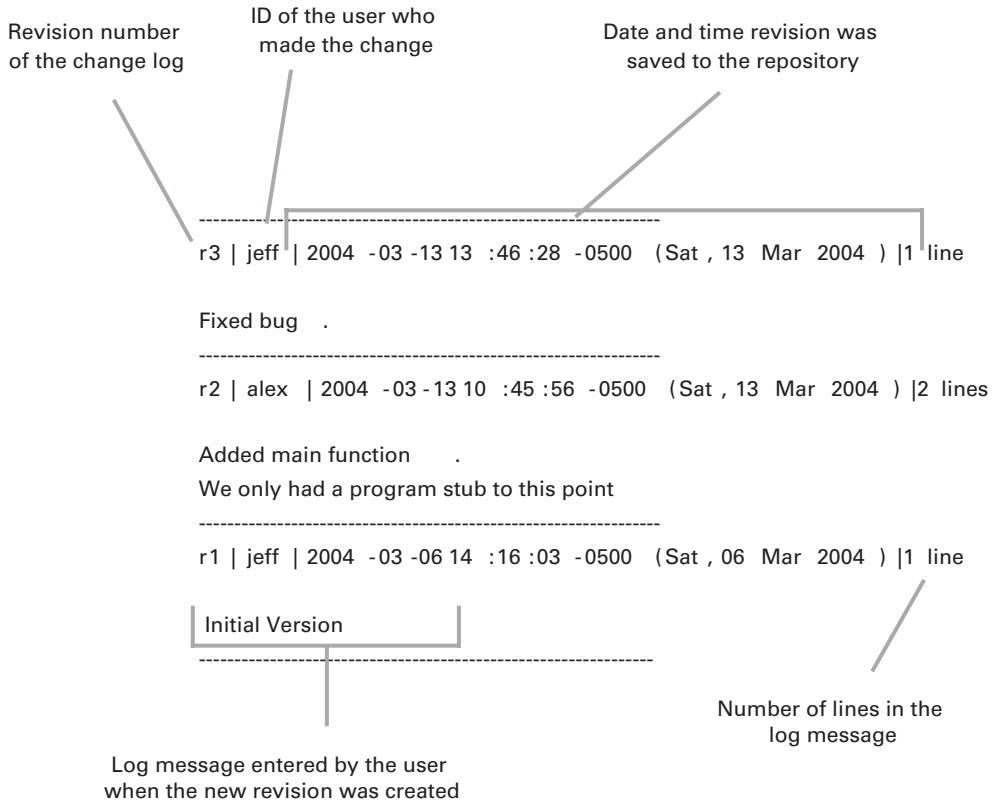


Figure 1.3 Change log information provided by subversion

Each revision's change log for the file `main.c` has the same set of information and is ordered by date in descending order. The top line of each change log is the system information. The line(s) following the system information is the log message. This is a description of the change for that revision, which the user adds as part of the commit. Since the log message is added by the user, the quality of the message will depend on that developer's attention to detail.

When developing software for the long term, it is important to be descriptive when writing the log message. It should be clear to all users why you are making the change and what the problem and fix are. To see how nondescript messages can be a problem, look at the following log message:

```
-----  
r3 | jeff | 2004-03-13 13:46:28 -0500 (Sat, 13 Mar 2004) | 1 line
```

```
Bug Fix  
-----
```

The log message simply says “bug fix.” While this may make sense to the developer at the time of the change, it will probably not make sense in the future and likely won’t help other developers who look at it. The only way to get any details about the change is to look at the content differences between revisions. This can be tedious and will not always give you the answers you are looking for. Even if you see the differences in the contents of the file, you still may not know why the change was made. Instead of just saying “bug fix,” consider using something like the following message:

```
-----  
r3 | jeff | 2004-03-13 13:46:28 -0500 (Sat, 13 Mar 2004) | 3 lines
```

```
Bug Fix number 2255. The application is supposed to print a message  
when it starts. This message did not print because it was directed  
to STDERR. Adjusted the startup println statement to STDOUT  
-----
```

This log message is clear. Now you can easily see what was fixed, which can drastically cut down your research time. If your development process uses a bug-tracking tool, you can include the tracker number in the log message for a quick and dirty integration. Change logs provide the basic information in a version control system, but you are not limited to this data. You can add your own customization and information to a version by using properties.

1.1.6 The role of properties

Today’s version control systems are much more than simple repositories for storing old versions of source code; they help you manage your development process. For example, most software you develop will have releases. You will want a way to associate a particular version of a file, or more likely multiple files, with a particular release of the entire application. So, for example, you can say that release 2.0 of the application uses revision 10 of the file `main.c`. In order to accommodate this, version control systems have something called *properties*, also known as *meta-data*. This is simply some user-defined piece of information that is associated with a file or revision of a file. If we stick with the analogy of a library, using properties

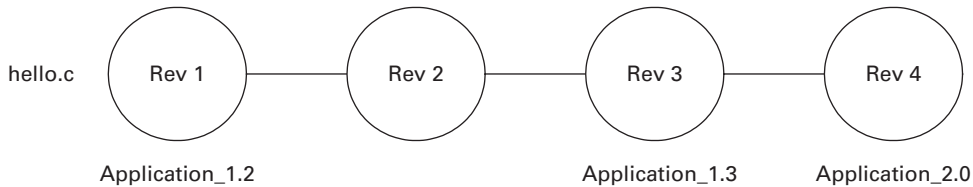


Figure 1.4 Tags in the version tree

to track releases of your software is similar to using a card catalog system. It allows you to easily track and find your files and see what state they are in.

So using properties, you can assign the name of the application release to a revision of the file to track the progress through the releases of your software. Figure 1.4 illustrates how this would look in a version tree for a file called `hello.c`.

The file `hello.c` has four revisions in the repository; three are associated with a particular release of your software. If version 2.0 of the application needs to be built, revision 4 of `hello.c` will be used. Versions not associated with a tag are either just checkpoints for a developer to save his work or not production worthy. You can use properties for any kind of custom information that is required in your development process.

1.2 30,000 foot view of Subversion

Now that you have an understanding of what most version control systems provide, we can better examine what is different about Subversion. Right off the bat, we know that Subversion uses current open source tools to fill needs that do not directly deal with version control. The following is a list of the key open source tools used:

- Berkeley DB is used for a back end, which provides an efficient storage facility in terms of speed and memory.
- The Apache Portable Runtime libraries are used, eliminating the need for operating system-specific code.
- The Apache HTTP server is used for the network protocol, which gives built-in security and web-browsing capabilities.

In addition to utilizing current technologies, Subversion has a few innovations that give it additional advantages. Let's look at some of them.

1.2.1 Atomic commits

“Atomic” means one global change to the repository, no matter how many files have changed. For example, say you make changes to five different files, instead of five new files being created in traditional systems, Subversion creates one new repository with all the changes. There is one revision number and one change log in an atomic commit, as opposed to one for each file. Don’t worry if this concept is not crystal clear yet; as you move through the book, it will make sense and the reasons why this was implemented will become obvious.

1.2.2 Everything is versioned

If you have been in software development for any length of time, you have probably been in a situation similar to this. You come in to work on a Monday morning to find your mailbox full of user complaints that a function in your software doesn’t work anymore, but you haven’t made any changes to the source code in two weeks. After some probing, you find out that this function gets run only annually, so the last time it worked was a year ago. Of course, there have been a dozen releases and patches to the software since last year. In order to see if a code change broke the function, you will need to travel back in time to re-create the repository at that point, build the application, and see if the problem exists. This ability to “go back in time” turns out to be a very powerful feature.

Any version control system can get the file contents back easily enough, but there may be more to it than that. For instance, what if some files were moved to different directories? Then you would need to fix the paths in your build scripts. If you are using tags or labels on the files to determine what was in production, there is no way track this unless you manually record it somehow. Subversion not only tracks changes to a file’s contents, it also versions directories and properties, which allows you to return the repository back to its original state, which consists of more than just file contents.

1.2.3 The three components of Subversion

Subversion consists of three components: the filesystem, the network, and the client. The filesystem component is the Subversion repository that stores the files and change logs. The client component is the set of libraries and command-line programs the user will run to access the repository. The network component can be used if the access to the repository is not local—it acts as an intermediary between the filesystem and the client. For local access, the client talks directly to the filesystem. Figure 1.5 shows the coupling points of the components.

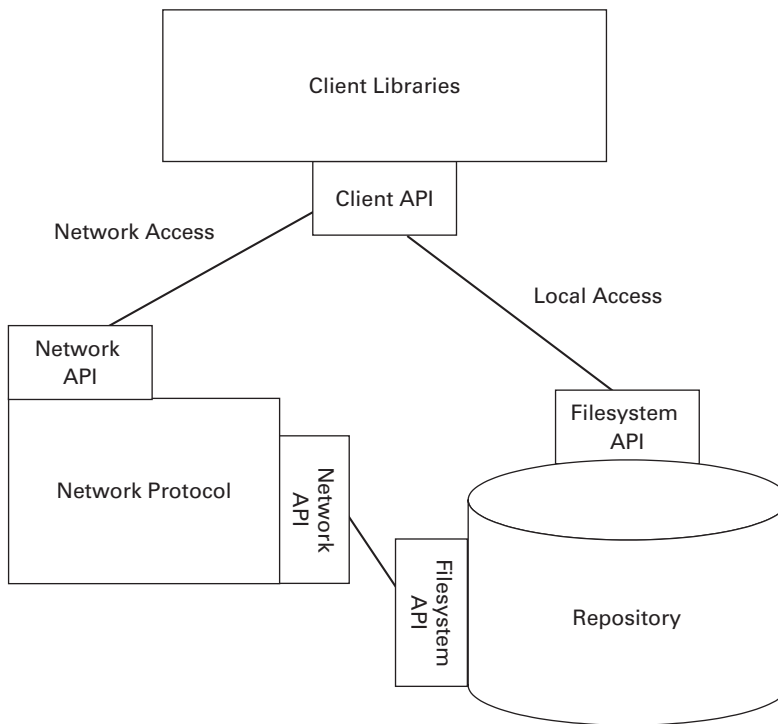


Figure 1.5 Subversion component coupling points

Each component is a standalone module and can be changed out without affecting the other two. So how does this modularity help you? First, it allows you to access the repository over the network or locally on the host. In addition, this design provides more third-party and vertical tools to sit on top of Subversion, and you can easily snap in different network protocols or client interfaces.

1.3 Revision numbers

Subversion has developed a new paradigm for identifying version numbers in a repository, one that is based on repository commits and not individual file changes. This is a little different than traditional version control systems, so it may take some getting used to.

1.3.1 The old way

In traditional version control systems, each file has its own revision number, which is an incremental decimal. So the revision numbers for a file would be 1.1, 1.2, 1.3, etc. There will not be any gaps because each time the file is changed, it is incremented off its own revision number (unless it is manually changed). To see how this can be a deficiency, consider two files: `main.c` is at revision 1.2 while `hello.c` is at revision 1.1. If a change is made to both files and committed to the repository, the output of the two logs would look something like this:

```
File: main.c
head: 1.3
-----
revision 1.3
date: 2004/03/09 18:23:14; author: alex; state: Exp; lines: +3 -0
Added author tags to source files that were missing it
-----
revision 1.2
date: 2004/03/09 18:21:19; author: jeff; state: Exp; lines: +1 -0
Added print statement so we know it started
-----
revision 1.1
date: 2004/03/09 18:16:45; author: jeff; state: Exp;
branches: 1.1.1;
Initial revision
-----

File: hello.c
head: 1.2
-----
revision 1.2
date: 2004/03/09 18:23:14; author: alex; state: Exp; lines: +4 -0
Added author tags to source files that were missing it
-----
revision 1.1
date: 2004/03/09 18:16:45; author: jeff; state: Exp;
branches: 1.1.1;
Initial revision
-----
```

Since the revision numbers are just sequential, `main.c` moves to 1.3 and `hello.c` is now at 1.2. There is nothing that ties the two revision numbers back to your change. The only way to tell which versions of files trace back to the same change is to look at the log file. This is fine if you are comparing files that you are aware of. What if a change caused your application to go bonkers, and you need to trace

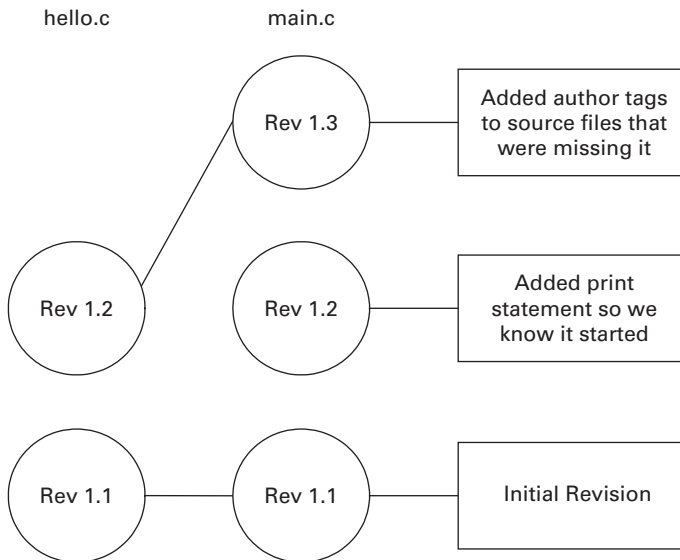


Figure 1.6
Version tree with traditional
version numbering

back to see what has been modified? To see how difficult this can be to trace, look at how the changes are correlated in the version tree in figure 1.6.

As you can see, it is difficult enough to trace changes with only two files. The same commit produced revision 1.2 of the file `hello.c` and revision 1.3 of `main.c`. In order to find all the files affected by a change, you will have to look through all your files for either a time stamp or specific log message. To get around this problem with most traditional version control systems, you will need to create a label and attach it to each new file revision. Unless you create a system to tag all previous revisions of a change, it will be difficult to back out of a change.

1.3.2 Subversion revision numbers

With the atomic commits, Subversion tracks changes using a per-commit basis, not per-file. Instead of each file maintaining a separate revision number, Subversion has one global revision number for the repository. The number starts at 0 when the repository is created and is incremented by a whole number each time a commit happens. Conceptually, each commit creates a new copy of the repository with the changes applied. Files that do not change on a commit simply have identical copies created in the new repository, although they are stored more efficiently than a complete copy. Let's

take a look at what the log files would look like in Subversion going through the steps in the previous example:

```
File: main.c
-----
r3 | alex | 2004-03-09 22:32:34 -0500 (Tue, 09 Mar 2004) | 1 line
Added author tags to source files that were missing it
-----
r2 | jeff | 2004-03-09 22:30:42 -0500 (Tue, 09 Mar 2004) | 1 line
Added print statement so we know it started
-----
r1 | jeff | 2004-03-09 22:28:18 -0500 (Tue, 09 Mar 2004) | 1 line
Imported Sources
-----

File: hello.c
-----
r3 | alex | 2004-03-09 22:32:34 -0500 (Tue, 09 Mar 2004) | 1 line
Added author tags to source files that were missing it
-----
r1 | jeff | 2004-03-09 22:28:18 -0500 (Tue, 09 Mar 2004) | 1 line
Imported Sources
-----
```

What should stand out when you compare the two log histories is that the file `hello.c` jumps from revision 1 to revision 3. Many people (including myself) cringe the first time they see this. Once you stop and think about it, this method makes more sense. When you commit to the repository, you are applying a set of common changes. If the revision number for each file is the same, it is easier to trace and back out if necessary.

This also makes it easier to check out the repository at a specific point. The reason for the ease is that all changes are at the same level in the version tree, as figure 1.7 shows.

Unlike the traditional numbering scheme we saw earlier, in Subversion all the files that were changed by the commit to add the author tags are part of revision 3.

When you check out a specific revision of the repository, you will get the highest revision of each file less than or equal to the number specified. So using the files `hello.c` and `main.c` from our previous example, you check out revision 2 of the repository. Since `main.c` has a revision 2, you will get that copy. Because `hello.c` does not have a revision 2, it will go down to revision 1. Since the commit is considered to be at a repository basis, the change log information is stored at the repository level instead of the file level. This means that instead of each file

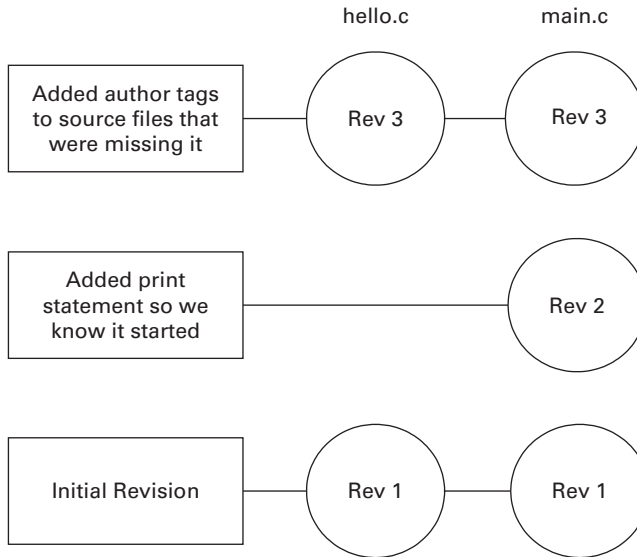


Figure 1.7
**Version tree with Subversion
 revision numbers**

revision containing the author, date, and change log, this information is kept only once in the repository. This actually allows Subversion to be more efficient in terms of storage. It also allows for easier maintenance; if you need to adjust the change log, you need to do so in only one place instead of every file.

1.4 Summary

We have discussed concepts such as atomic commits and the copy-modify-merge model at a high level, but these may not be perfectly clear to you yet. As you read through this book and explore the commands, these concepts will become second nature. Just remember that it will be up to you to keep your working copy in sync with the repository, with both your changes and those of other developers.

You are now ready to dive into Subversion! Whether you are a novice to traditional version control systems or a seasoned veteran, the best way to learn how to use this tool is to walk through the steps as we discuss them. Don't be afraid to experiment and try different things. If you create a test repository, the worst case would be that you blow it away and have to start over. If you are brand new to version control systems and are feeling a little wary, don't worry. You can easily set up and destroy test repositories to get the feel for them. We will go through Subversion sequentially, starting with creating a repository.

Subversion IN ACTION

Jeffrey Machols

A new-generation version control tool, Subversion is replacing the current open source standard, CVS. With Subversion's control components you can simplify and streamline the management of your code way beyond what's possible with CVS. For example, with just one powerful feature, Subversion's atomic commit, you can easily track and roll back a set of changes.

Subversion in Action introduces you to Subversion and the concepts of version control. Using production-quality examples it teaches you how Subversion features can be customized and combined to effectively deal with your day-to-day source control problems. You'll learn how to do practical things you cannot do with CVS, like seamlessly renaming and moving files.

The book covers branching and repository control, access control, and much more. It is written not just for release engineers, but also for developers, configuration managers, and system administrators.

What's Inside

- Integrate Subversion into your development environment
- Repository creation, backup, and options
- Svnadmin and svnlook client interfaces
- Change logs and comparing versions
- Advanced administration and configuration commands
- Lifecycle development with Subversion

A system administrator and developer with over ten years of experience, **Jeffrey Machols** is a co-founder of the Apache Directory Project and an early adopter of Subversion. He lives in Jacksonville, Florida.

“... the best book on Subversion.”

—Michael Oliver
CTO, Matrix Intermedia Inc.

“... crammed with easy-to-follow examples on how to use Subversion.”

—Alex Karasulu, Apache Directory Project co-founder

“... an indispensable tool.”

—Mark Maimone
Rochester Institute of Technology

“Without this book, any environment using Subversion is incomplete.”

—Paul Bonkowski
Senior Architect, LB Industries

“... gets you up and running quickly.”

—Lübbe Onken
RA Consulting GmbH, and TortoiseSVN Developer



www.manning.com/machols



ISBN 1-932394-36-2