

It's Next-Generation Open Source Version Control

Version Control with

Subversion



O'REILLY®

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

Basic Concepts

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

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

The Repository

Subversion is a centralized system for sharing information. At its core is a repository, which is a central store of data. The repository stores information in the form of a *filesystem tree*—a typical hierarchy of files and directories. Any number of *clients* connect to the repository, and then read or write to these files. By writing data, a client makes the information available to others; by reading data, the client receives information from others. Figure 2-1 illustrates this. Why is this interesting? So far,

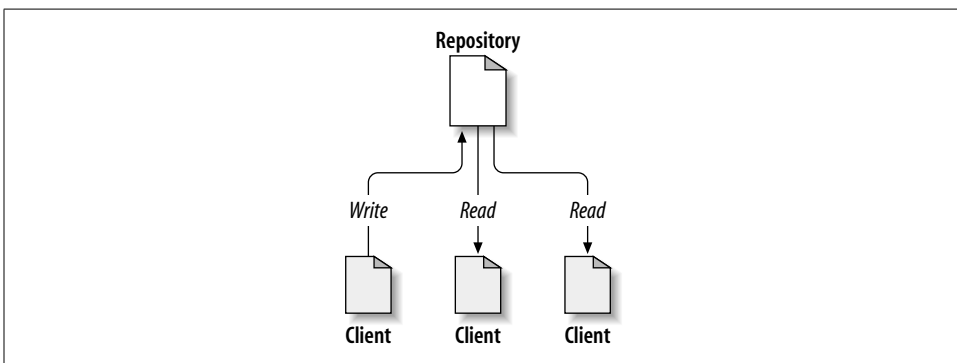


Figure 2-1. A typical client/server system

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

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

Versioning Models

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

The Problem of File Sharing

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

Consider the scenario shown in Figure 2-2. Suppose we have two co-workers, Harry and Sally. They each decide to edit the same repository file at the same time. If Harry saves his changes to the repository first, then it's possible that (a few moments later) Sally could accidentally overwrite them with her own new version of the file. While Harry's version of the file won't be lost forever (because the system remembers every change), any changes Harry made *won't* be present in Sally's newer version of the file, because she never saw Harry's changes to begin with. Harry's work is still effectively lost—or at least missing from the latest version of the file—and probably by accident. This is definitely a situation we want to avoid!

The Lock-Modify-Unlock Solution

Many version control systems use a *lock-modify-unlock* model to address this problem. In such a system, the repository allows only one person to change a file at a time. First Harry must lock the file before he can begin making changes to it. Locking a file is a lot like borrowing a book from the library; if Harry has locked a file, then Sally cannot make any changes to it. If she tries to lock the file, the repository will deny the request. All she can do is read the file, and wait for Harry to finish his

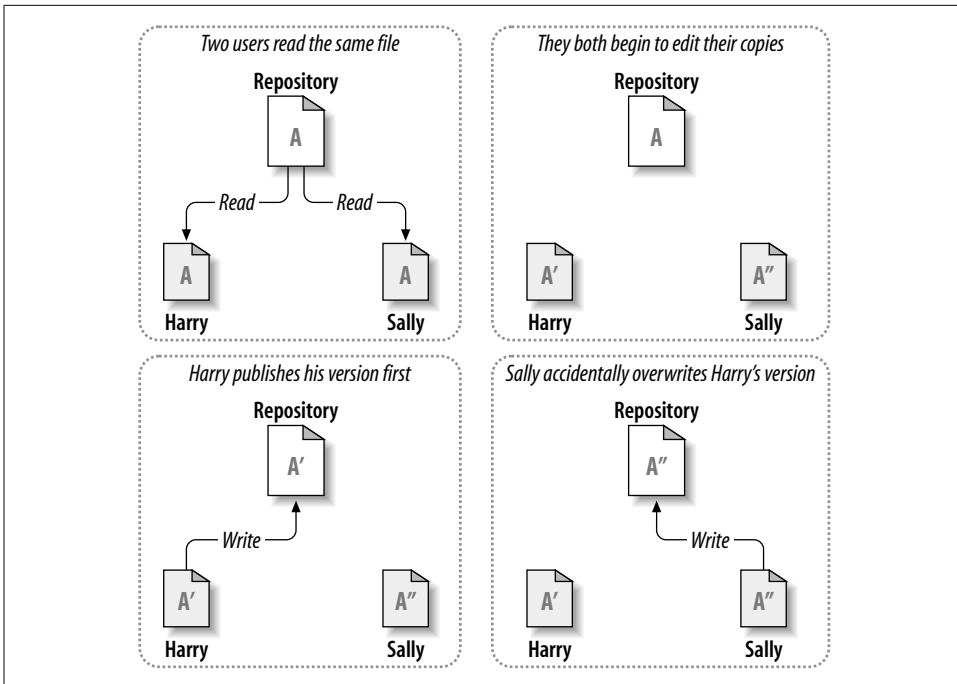


Figure 2-2. *The problem to avoid*

changes and release his lock. After Harry unlocks the file, his turn is over, and now Sally can take her turn by locking and editing. Figure 2-3 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 locks 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.

Pretend that Harry locks and edits file A, while Sally simultaneously locks and edits file B. But suppose that A and B depend on one another, and the changes

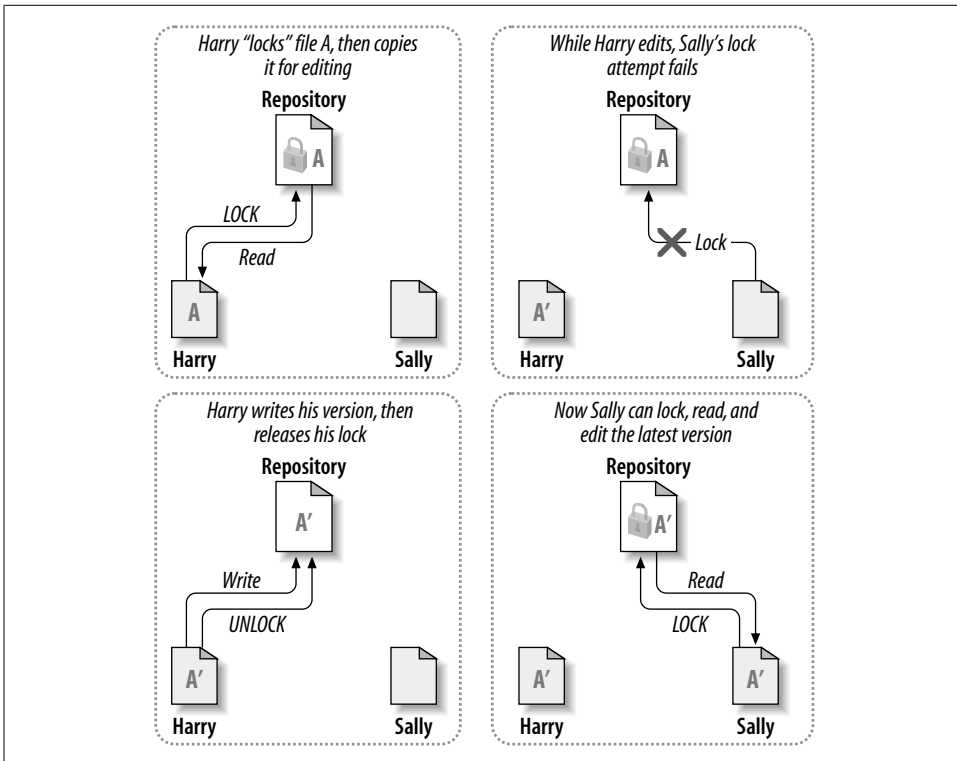


Figure 2-3. The lock-modify-unlock solution

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 inhibits them from discussing their incompatible changes early on.

The Copy-Modify-Merge Solution

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

Here's an example. Say that Harry and Sally each create working copies of the same project, copied from the repository. They work concurrently, and make changes to the same file A within their copies. Sally saves her changes to the repository first.

When Harry attempts to save his changes later, the repository informs him that his file A is *out-of-date*. In other words, that file A in the repository has somehow changed since he last copied it. So Harry asks his client to *merge* any new changes from the repository into his working copy of file A. Chances are that Sally's changes don't overlap with his own; so once he has both sets of changes integrated, he saves his working copy back to the repository. Figures 2-4 and 2-5 show this process.

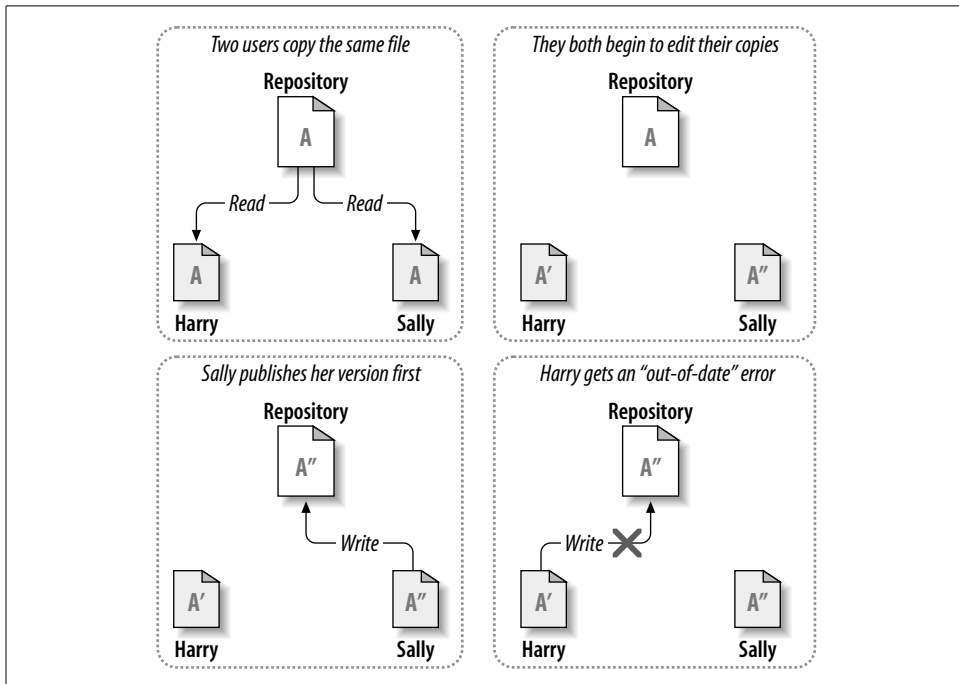


Figure 2-4. The copy-modify-merge solution

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.

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

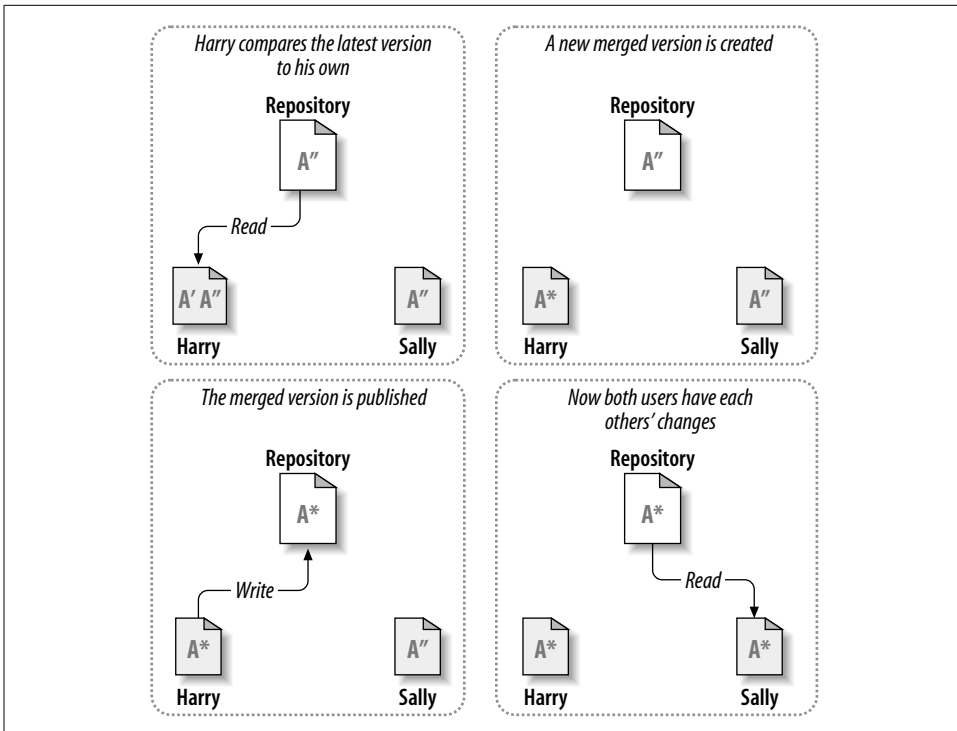


Figure 2-5. The copy-modify-merge solution (continued)

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

Subversion in Action

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

Working Copies

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

A Subversion working copy is an ordinary directory tree on your local system, containing a collection of files. You can edit these files however you wish, and if they're source code files, you can compile your program from them in the usual way. Your

working copy is your own private work area: Subversion will never incorporate other people's changes, nor make your own changes available to others, until you explicitly tell it to do so.

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

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

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

For example, suppose you have a repository that contains two software projects, *paint* and *calc*. Each project lives in its own top-level directory, as shown in Figure 2-6.

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
A calc/Makefile
A calc/integer.c
A calc/button.c

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

The list of letter A's indicates that Subversion is adding a number of items to your working copy. You now have a personal copy of the repository's */calc* directory, with one additional entry—*.svn*—that 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 modification date and original contents, Subversion can tell that you've changed the file. However, Subversion does not make your changes public until you explicitly tell it to do so. The act of publishing your changes is more commonly known as *committing* (or *checking in*) changes to the repository.

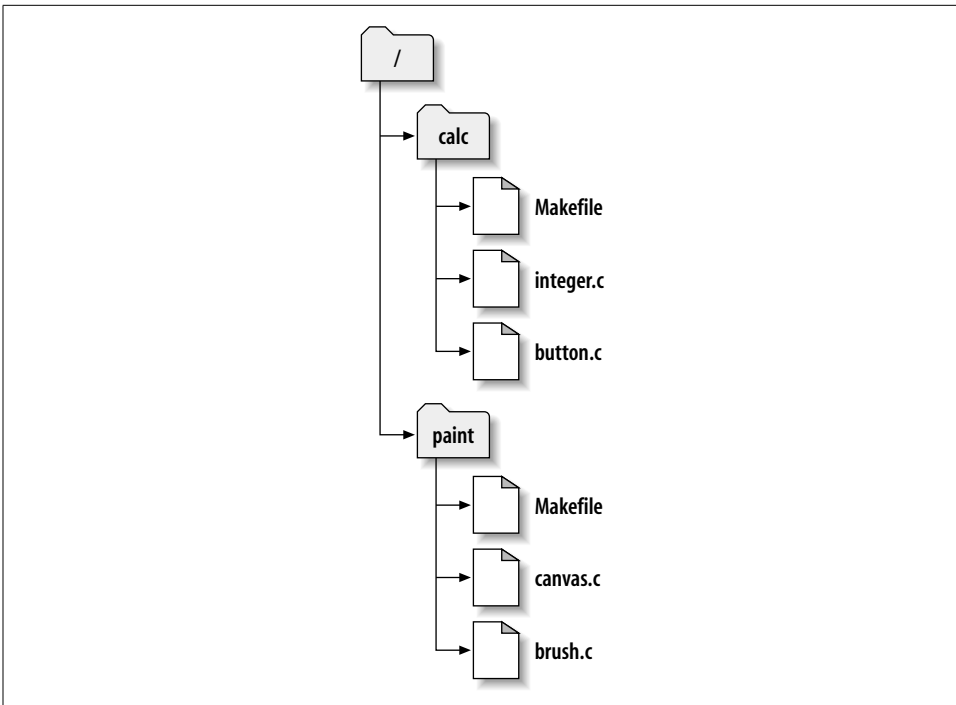


Figure 2-6. The repository's filesystem

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

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

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

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

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

```
$ pwd
/home/sally/calc
```

Repository URLs

Subversion repositories can be accessed through many different methods—on local disk, or through various network protocols. A repository location, however, is always a URL. Table 2-1 describes how different URL schemas map to the available access methods.

For the most part, Subversion’s URLs use the standard syntax, allowing for server names and port numbers to be specified as part of the URL. Remember that the `file:` access method is valid only for locations on the same server as the client—in fact, in accordance with convention, the server name portion of the URL needs to be either absent or `localhost`:

```
$ svn checkout file:///path/to/repos
...
$ svn checkout file://localhost/path/to/repos
...
```

Also, users of the `file:` scheme on Windows platforms 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:/path/to/repos
...
C:\> svn checkout "file:///X|/path/to/repos"
...
```

In the second syntax, you need to quote the URL so that the vertical bar character is not interpreted as a pipe.

Note that a URL uses ordinary slashes even though the native (non-URL) form of a path on Windows uses backslashes.

Table 2-1. Repository access URLs

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

```
$ ls -a
.svn/ Makefile integer.c button.c
```

```
$ svn update
U button.c
```

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

Revisions

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

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

Each time the repository accepts a commit, this creates a new state of the filesystem tree, called a *revision*. Each revision is assigned a unique natural number, one greater than the number of the previous revision. The initial revision of a freshly created repository is numbered 0, and consists of nothing but an empty root directory.

Figure 2-7 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 each commit.

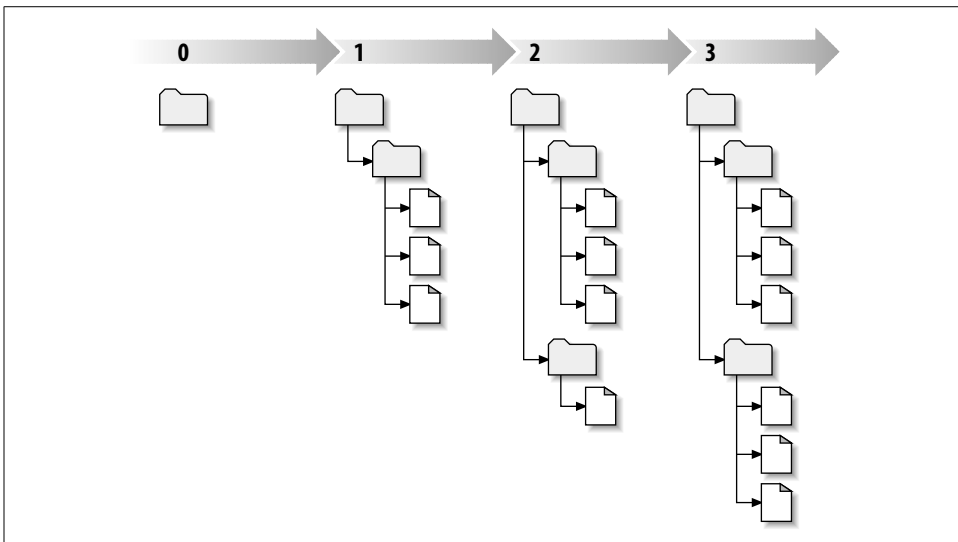


Figure 2-7. The repository

Global Revision Numbers

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

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

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

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

Sally's changes to *integer.c* appears in your working copy, and your change is still present in *button.c*. In this example, the text of *Makefile* is identical in revisions 4, 5, and 6, but Subversion marks your working copy of *Makefile* with revision 6 to indicate that it is still current. Thus, after you do a clean update at the top of your working copy, it generally corresponds to exactly one revision in the repository.

How Working Copies Track the Repository

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

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

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

Unchanged, and current

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

Locally changed, and current

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

Unchanged, and out-of-date

The file has not been changed in the working directory, but it has been changed in the repository. The file should eventually be updated, to make it current with the public revision. An `svn commit` of the file will do nothing, and an `svn update` of the file will fold the latest changes into your working copy.

Locally changed, and out-of-date

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

This may sound like a lot to keep track of, but the `svn status` command will show you the state of any item in your working copy. For more information on that command, see “`svn status`” in Chapter 3.”

The Limitations of Mixed Revisions

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 mixed revision numbers.

At first, it may not be entirely clear why this sort of flexibility is considered a feature, and not a liability. After completing a commit to the repository, the freshly committed files and directories are at a more recent working revision than the rest of the working copy. It looks like a bit of a mess. As demonstrated earlier, the working copy can always be brought to a single working revision by running `svn update`. Why would someone *deliberately* want a mixture of working revisions?

Assuming your project is sufficiently complex, you'll discover that it's sometimes nice to forcibly backdate portions of your working copy to an earlier revision; you'll learn how to do that in Chapter 3. Perhaps you'd like to test an earlier version of a submodule, contained in a subdirectory, or perhaps you'd like to examine a number of previous versions of a file in the context of the latest tree.

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

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

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

Summary

We 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 gave 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 jump into the next chapter, which is a detailed tour of Subversion's commands and features.