CHAPTER 1

Introducing Subversion

So you're holding in your hands a copy of *Practical Subversion*, and that suggests that you're curious about the Subversion version control system. Before diving into the details of how to use Subversion, you'll need to know a little about what *version control* is and why you probably should be using it, and a bit about Subversion itself. This chapter will cover all of that.

What Is Version Control?

Any sufficiently large project eventually hits a point at which communication between the participants becomes a bottleneck. Making sure everyone has the latest version of some crucial bit of information, avoiding duplication of effort, and ensuring that one person doesn't inadvertently overwrite the work of another starts out looking easy, but as the scope of a project grows and the number of people involved grows with it, what once seemed simple becomes more and more complex, and as a result takes more time and effort.

In the software development world, one of the ways to manage this complexity is through the use of version control systems.

Version control provides a way to manage the complexity of working with a team of people on a specific project. It allows you to store the ongoing results of the project (in software development, that would be the source code and documentation for the program being developed) in a central location so that the various members of the team always have access to the latest versions of critical files. It automates the process of updating your local working copy of the project to incorporate the latest changes from your teammates, and it allows you to push your changes upstream into the repository so that other users can have access to them. Finally, and perhaps most importantly, it provides you with historical data regarding the progress of the project, tracking each and every change that was made in the process of its creation. This might not seem important at first glance, but eventually in the life cycle of a project you'll want to go back and see how and why the project evolved into its current form, either to understand the source of a bug, to retrieve some bit of work that was inadvertently removed from the current version, or perhaps to extract a particular change so that it can be applied to another version of the project.

In general, using a version control system as part of your project works as follows: First a space for the project is created in the repository and the initial version is checked in. Then each person working on the project checks out a local copy, and as someone makes changes, he pushes them back into the repository. The repository keeps track of the history of the project, storing each change that's submitted. From time to time each user updates his local copy,

and each of the changes other users have made to the repository is applied to the user's local copy. Different systems use different techniques to ensure that changes from one user don't conflict with the work other users have in progress, so you don't have to worry about two people changing the same part of a project and accidentally overwriting each other's work. Because the full history of each change is kept, usually with additional comments from each change's author explaining the change's purpose, it's easy to go back in time and figure out why any given change was made, which becomes quite valuable when debugging problems in the future.

Each of these pieces of functionality is useful, but the combination becomes invaluable, and once you've used a version control system as part of your job, it's unlikely that you'll ever go back to working without one.

The Concurrent Versioning System

Within the open source community, and among a great many other software developers worldwide, the Concurrent Versioning System (CVS) has long been the accepted standard in version control software. Its open access and nonlocking mode of operation has enabled countless developers to collaborate on projects both large and small.

CVS originated as a set of scripts wrapped around an older version control system, the Revision Control System (RCS). While none of the code from the original shell script form of CVS remains in its current incarnation, many of the algorithms used for conflict resolution in CVS originated in RCS.

You see, unlike RCS, and many other version control systems before it, CVS is generally *nonlocking*. This means that when a user is editing a file, there's nothing to prevent another user from also changing that file at the same time. This might seem counterintuitive at first, because there's nothing preventing the two users from making conflicting changes, but in practice it turns out to work quite well. Even when two people make changes to the same file, those changes can usually be merged together automatically. The first user who checks in her change simply goes on without knowing that another user is editing the file, and when the second user tries to check in his change, he's informed that his copy is out-of-date. He updates his local copy, and CVS tries to merge his changes in with the new change that was committed to the repository. Assuming the changes don't conflict, that's all there is to it. The user verifies that his changes are still correct and tries to check it in again; and this time it succeeds. If there's a conflict between the two changes—because the users edited the same part of the file, for example—the second user is presented with all three versions (the original, his modified version, and the new version in the repository), and he can manually resolve the conflict.

This type of workflow seems rather chaotic at first, with nothing to prevent two users from modifying the same file at the same time. In reality, though, it works very well, especially in situations in which a development team is distributed across a country, or even across the world. Because of this, and the fact that it's freely available, CVS has been adopted as the de facto standard in version control for open source software development, and for much commercial software development as well.

Despite its widespread adoption and use, CVS is far from perfect, as anyone who has used it for any length of time will likely be happy to tell you. Its user interface can be idiosyncratic; its implementation is often inefficient; and more often than not, attempts to improve CVS are hamstrung by overwhelming amounts of historical baggage. There's only so much that can be done to improve a system that was built on top of a shaky foundation, and CVS's RCS roots are in many ways rather unstable.

Because CVS was built on top of RCS, it inherited a number of problems. The RCS file format is designed to track changes on a per-file basis; it has no concept of directories; and any change that includes a file has to modify that file's associated RCS file. This means that things such as marking all the files in your project as belonging to a particular version of your software (commonly referred to as *tagging*) or starting a new line of development so that one version of your project can evolve separately from the mainline development version (commonly known as *branching*) take time and space proportional to the number of files they affect, which can become problematic as time goes on. Similarly, because changes are made on a per-file basis, it's possible for half of a commit to succeed but for the remainder to fail, say because the user's network connection died partway through. Worse yet, even if everything goes fine during the commit, it can take a long time; meanwhile, it's possible for another user to update and get only part of the change. The result of both situations is an inconsistent set of files, which can be quite annoying for users.

Yet even with all of CVS's shortcomings, there's never been an acceptable replacement that could really fill its shoes. Until now.

What Is Subversion?

In early 2000, CollabNet Inc. (`http://www.collab.net/`) decided that its collaborative software suite, SourceCast, required a version control system that could replace CVS. Although CVS had many useful features, it was clear that CollabNet's users were running up against its limitations, and the company's solution was to create a new open source project with the goal of creating a CVS replacement. CollabNet contacted Karl Fogel, the author of a well-known book on CVS, and asked him to work on the project. Fogel and his friend Jim Blandy, an engineer at Red Hat Inc., had been discussing a design for a new version control system, and Blandy had even come up with a name, Subversion. Fogel went to work for CollabNet, and Blandy convinced Red Hat to let him work on the project full time. With the help of the higher-ups at CollabNet, many of whom are well-known open source developers in their own right, Subversion began to attract a community of developers interested in replacing CVS with something similar, but better.

Subversion became *self-hosting* on August 31, 2001, which means that the Subversion developers stopped using CVS to hold Subversion's own code and started using Subversion instead. A series of releases followed, culminating in a beta release on December 19, 2003, and finally version 1.0 on February 23, 2004. It was a long road, but the Subversion developers had created something wonderful. Since that time, Subversion has had three more major version releases, with the current version being 1.3.2, and 1.4 expected soon.

Subversion retains many of the general concepts of CVS and corrects most of the egregious problems. The Subversion developers, many of whom are longtime CVS users and developers, have taken great care to preserve the portions of CVS that are most useful, while simultaneously improving upon it in countless ways. Specifically, Subversion provides the following big-ticket improvements over CVS, described in the next section.

Versioned Directories, Renames, and File Metadata

Whereas CVS works in terms of individual files, Subversion versions entire trees, which allows it to track historical data that CVS simply can't store. Many CVS users will admit to being shocked when they first realized there was no simple way to rename a file or delete a directory, or that they couldn't have an empty directory in their CVS repository.

When we say that files and directories have *metadata*, it means that arbitrary bits of data about each file or directory can be stored by Subversion. This data is used by Subversion itself for keeping track of things such as the line-ending style used by the file, the fact that the file is executable, or the keywords that Subversion should expand within the file. Additionally, you can create your own bits of data to associate with files and directories. Perhaps you want to track the last time the translation of a particular bit of documentation was updated, or who holds the copyright on the file, or any other piece of information you can think of, really. The possibilities are limitless.

To learn how to manage directories, files, and their associated metadata, see Chapter 2.

Atomic Changes to the Repository

With truly atomic changes, all parts of a given change go into the Subversion repository or none of them do. For example, if you run `svn commit foo.c bar.c`, your changes to both `foo.c` and `bar.c` will either make it into the repository together or not at all.

This might not seem important, but imagine a change that affects hundreds of files. The process of checking in the change might take some time, and in systems without atomic changes, it's possible that someone updating her checked-out copy while the change is happening would get part of the changes, but not all of them. It's easy to see how this could result in a set of files that aren't internally consistent, and thus don't work.

In Subversion, there's no way you can update your checked-out copy of the tree at the wrong time and receive half of a large change, as can often happen in systems such as CVS. We discuss this in more detail in Chapter 2.

Branching and Tagging

There's no reason for creating a new branch or tagging a release to take linear time relative to the number of files involved, as happens in CVS. Instead, in Subversion, branching and tagging take a constant (and small) amount of time and disk space, thanks to the software's advanced repository design. For more information on using branches, see Chapters 2 and 6.

Client/Server Application Design

Software developers will appreciate that Subversion has been designed from the start to meet their many requirements, rather than evolving to meet them, as CVS did. Subversion has been carefully designed as a native client/server system, with well-defined boundaries between its constituent libraries, making it much more approachable for a developer looking to add a feature or fix a bug. For more information on Subversion's network support, see Chapter 3.

Saving Bandwidth

If the Subversion client has access to both the original version of a file you checked out from the repository and the new version you're trying to check in, it can get away with only sending the difference between the two files to the server, which takes considerably less bandwidth than sending the entire new file.

Disconnected Operations Support

Network bandwidth is generally cheap, but disk space has become cheaper, so Subversion is optimized to avoid network traffic whenever possible, caching data on the client side and allowing many actions to avoid using the network at all.

We haven't devoted time in any specific part of the book to discuss this in detail, but keep it in mind as you read through these first few chapters. You're likely to see several places where design decisions are made in such a way as to use extra disk space to minimize network traffic.

Well-Supported Binary Files

Experience has proven that CVS handles binary files poorly (at best). Subversion improves on this in two ways. First, the client is designed to make it difficult for a user to make an error that results in the destruction of his file. (In CVS, it's all too easy to have end-of-line conversion or keyword translation performed on a binary file, rendering it useless—or worse, unrecoverable.) Second, the Subversion repository uses an efficient binary diff algorithm, so that storing each revision of a binary file separately (as CVS does) isn't necessary. For information about Subversion's handling of binary files, see Chapter 2 (specifically the "Properties" section).

Sharing a Repository Over a Network

If you're looking for simplicity, a custom TCP has been written, with a simple server process. This protocol can be tunneled over SSH if necessary, leveraging the existing security infrastructure that's evolved to support CVS. If you're looking for even more flexibility, Subversion can use a WebDAV-based protocol, sending HTTP requests over the wire to the most time-tested server process around, Apache. The Apache-based server allows you to make use of a giant pool of existing Apache modules, many of which can provide Subversion with capabilities far in excess of what Subversion's developers could provide on their own. For example, you can use an Apache module to allow your Subversion server to store its list of users and passwords in a MySQL database rather than a file on disk, or to compress the data that's transmitted over the network via the `zlib` library.

For more information on Subversion's networking options, see Chapters 3 and 5.

Workflow Similar to CVS

Perhaps most important for CVS users everywhere, Subversion is similar enough to CVS that you can quickly jump to using Subversion with a minimum amount of difficulty. The similarities between Subversion and CVS, combined with Subversion's consistent command-line user interface, make switching to Subversion considerably more palatable to those who have been dissatisfied with CVS in the past.

Like CVS before it, Subversion is an open source project. While the project is sponsored by CollabNet, a commercial company, the code for Subversion is available under an Apache-like license, free for any and all to use for any purpose.

Since Subversion's beginnings in early 2000, a thriving community of developers has grown up around it, all working to produce a system they can readily use in their day-to-day work. To learn more about participating in the Subversion community, see the `README` in the Subversion source distribution, and the `HACKING` file online at `http://subversion.tigris.org/hacking.html`, or visit the project's web page at `http://subversion.tigris.org/`.

Key Technologies

Subversion can make use of a number of technologies, such as the Apache web server and Berkeley DB, in order to provide option features. Whether you use them or not, you'll be able to better understand the rest of the book if you know a little bit about the technologies Subversion is built on.

Apache Web Server and WebDAV

The Apache HTTPD server (`http://httpd.apache.org/`) is the Internet's most popular web server. At any given time, between 60% and 70% of the world's web pages are served up by some version of Apache (see `http://www.netcraft.com/` for statistics on HTTPD server use).

WebDAV (`http://www.webdav.org/`) is a set of extensions to HTTP (the underlying protocol used by web browsers and web servers) that allow users to edit and manage files on remote web servers.

If you install a recent version of Apache and configure `mod_dav`, the module that provides Apache's WebDAV support, you can use a WebDAV client to manage the content served up by your server, rather than just reading it in a browser. A number of common operating systems have support for WebDAV built right in. For example, Mac OS X's Finder allows you to mount a WebDAV share right on your desktop as a network drive. Windows XP's Web Folders allow you to do essentially the same thing, as does GNOME's Nautilus file manager.

Subversion's `mod_dav_svn` Apache module and `libsvn_ra_dav` library make use of a subset of WebDAV and an extension to WebDAV known as DeltaV to allow a Subversion client to access a Subversion repository through an Apache web server. See Chapters 3 and 5 for more information on using `mod_dav_svn` and Apache with Subversion.

Berkeley DB

Subversion has two back ends in which it can store data. One of these is a filesystem-based back end, which is now the default, which will work best for the majority of people. However, some people feel safer if the data is stored in a real database, in which case Subversion provides a back end where the data is stored in a database using Berkeley DB.

Berkeley DB is an embedded database. Everything that's required for Subversion to access Berkeley DB, if you choose to use it, is built right in to Subversion itself.

We will cover which back end you should choose for your particular needs in Chapter 3.

Obtaining Subversion

Like many open source projects, Subversion is readily available in source code form. Tarballs of each release are made available from `http://subversion.tigris.org/` in the Documents & Files section of the site. Once you've downloaded the latest release tarball, building a Subversion client is relatively straightforward. On the other hand, if you're not interested in compiling Subversion yourself, precompiled packages and binaries are available for most major platforms.

Obtaining Precompiled Binaries

If you're not comfortable building Subversion from source or you don't have access to the tools required to build it, that's not a problem, as the developer community provides prepackaged releases for a wide variety of operating systems. Simply check `http://subversion.tigris.org/project_packages.html` for your operating system of choice and follow your platform's standard procedure for installing third-party software. Then you should be all set.

Building Subversion on Unix Systems

Before we describe the process of building and installing Subversion, it's worth pointing out that the `INSTALL` file in the top level of the release contains up-to-the-minute information on installing Subversion, which could be more up-to-date than the instructions presented here.

Once you've downloaded a tarball from `http://subversion.tigris.org/`, getting a basic client compiled is as simple as unpacking the tarball, running the `configure` script, and running `make` and `make install`, as the following shows:

```
$ curl -O http://subversion.tigris.org/tarballs/subversion-1.3.1.tar.gz
  % Total    % Received % Xferd  Average Speed          Time             Curr.
                                 Dload  Upload Total    Current  Left    Speed
100 8572k  100 8572k    0     0   251k      0  0:00:28  0:00:28  0:00:00  244k
$ tar zxf subversion-1.3.1.tar.gz
$ cd subversion-1.3.1
$ ./configure --prefix=/opt/packages/subversion
[ ... lots of output ... ]
$ make
[ ... even more output ... ]
# make install
[ ... still more output ... ]
$ /opt/packages/subversion/bin/svn
Type 'svn help' for usage.
$
```

If you just want to access an existing repository hosted on another machine, these are the basic steps you'll need to take. If you want to create your own repository, set up a server so that other people can access your repository, and run through most of the examples in the book yourself, you may need to do a few other things. Specifically, if you want to use `mod_dav_svn`, the Apache module that allows you to use an Apache HTTPD server as your Subversion server, you'll have to tell `configure` how to find your HTTPD install.

To create and access a Subversion repository using the database back end, you'll need to install an appropriate version of Berkeley DB (available from `http://sleepycat.com/download/index.shtml`) and ensure that Subversion can find it by passing a few options to the `configure` script.

Subversion will work with any of the earlier 4.x series of Berkeley DB releases. In particular, the absolute newest version, 4.4, is not supported by any Subversion that has been released. That said, developers have encountered various issues when using the 4.1.x versions, so you're almost certainly better off using either a 4.2.x or a 4.3.x version of Berkeley DB. Furthermore, Berkeley DB version 4.2.52 and later support additional features (such as automatic cleanup of old log files) that are nice to have, and the Subversion team has seen the best stability and

performance with the 4.3.x series of releases, so if it is at all possible you should probably use a recent version from the 4.3.x series. If that isn't possible for some reason, the next best option is 4.2.x. The 4.1.x releases are best avoided.

Besides Berkeley DB, Subversion makes use of the Apache Portable Runtime (APR) to abstract away the various platform-specific details of the operating systems it runs on, and it additionally makes use of the APR-Util library, a set of useful bits and pieces of code that are built on top of APR, which makes use of Berkeley DB. To lower the odds of shooting yourself in the foot by linking against multiple conflicting versions of Berkeley DB, Subversion simply makes use of whatever version of Berkeley DB that APR-Util uses. If your version of Berkeley DB is installed in a fairly standard place, APR-Util's `configure` script will most likely be able to find it. To play it safe, though, you should probably ensure that APR-Util links against the correct version of Berkeley DB. There are two options you can pass to APR-Util's `configure` script to make this happen. First, the `--with-dbm=db4` option will keep APR-Util from picking up earlier versions of Berkeley DB. Second, the `--with-berkeley-db=/path/to/berkeley/db` option will force APR-Util to use a version of Berkeley DB installed in the directory `/path/to/berkeley/db`.

■ **Note** Of course, you should replace `/path/to/berkeley/db` with the path to your Berkeley DB install, which is probably something like `/usr/local or /usr/local/BerkeleyDB.4.2`.

If you're using the versions of APR and APR-Util that are bundled with Subversion, you can ensure that these options are passed to APR-Util's `configure` script by passing them to Subversion's `configure` script. Similarly, if you're using the versions of APR and APR-Util that are compiled into the Apache HTTPD, you can simply pass those options to the HTTPD `configure` script. Sadly, one pitfall when building `mod_dav_svn` is that Subversion *must* be linked with the same versions of APR and APR-Util that Apache is. Subversion's `configure` script should take care of this for you, assuming you passed it the `--with-apxs` flag, but you can probably still get yourself into trouble with the `--with-apr` and `--with-apr-util` flags; so if you're building `mod_dav_svn` you should avoid those two flags. If you use the bundled versions, the whole thing looks something like this:

```
$ tar zxvf subversion-1.3.1.tar.gz
$ cd subversion-1.3.1
$ ./configure --with-dbm=db4 --with-berkeley-db=/opt/packages/bdb43
[ ... lots of output ... ]
$ make && make install
[ ... still more output ... ]
$
```

When using Apache with Subversion, there's another case where you may need to tell Subversion's `configure` script what to do. If Apache isn't installed in a standard place, you'll have to pass the `--with-apxs=/path/to/your/apxs` flag to Subversion's `configure` script. This will allow `configure` to use the `apxs` script that comes with Apache to figure out how to use your Apache installation when building Subversion.

Here's something else you should know about using Subversion with Apache: You absolutely must use a recent version of Apache 2.x—the older Apache 1.3.x simply will not

work. If you have Apache 1.3.x installed, you can still use Subversion, but you'll have to install Apache 2.x to use `mod_dav_svn` as your server.

Last in the list of pitfalls related to building Subversion with Apache support is that Apache needs to be compiled with a few special flags. First of all, `mod_dav_svn` depends on `mod_dav`, so you need to pass `--enable-dav` to Apache's `configure`. Second, because `mod_dav_svn` is normally compiled as a loadable module, you need to compile loadable module support into Apache. To do that, you need to pass `--enable-so` to Apache's `configure`, as shown in the following code:

```
$ tar zxvf httpd-2.0.55.tar.gz
$ cd httpd-2.0.55
$ ./configure --prefix=/opt/packages/apache2 \
              --with-dbm=db4 \
              --with-berkeley-db=/opt/packages/bdb43 \
              --enable-dav \
              --enable-so
[ ... lots of output ... ]
$ make && make install
[ ... lots of output ... ]
$ cd ..
$ tar zxvf subversion-1.3.1.tar.gz
$ cd subversion-1.3.1
$ ./configure --with-apxs=/opt/packages/apache2/bin/apxs
[ ... lots of output ... ]
$ make && make install
[ ... a lot more output ... ]
$
```

Table 1-1 summarizes the most commonly used Subversion `configure` script flags.

Table 1-1. *Useful configure Flags for Subversion*

Flag	Argument	Notes
`--prefix`	Path you want Subversion installed into	
`--with-dbm`	Version of Berkeley DB you want to find (e.g., db4)	Also applies to Apache and APR-Util configure scripts
`--with-berkeley-db`	Path to Berkeley DB install directory	Also applies to Apache and APR-Util configure scripts
`--with-apxs`	Path to apxs binary for installed Apache	
`--with-neon`	Path to installed Neon library	Only needed if you don't want to use the bundled version of Neon
`--with-apr`	Path to installed APR library	Only needed if you don't want to use the bundled version of APR
`--with-apr-util`	Path to installed APR-Util library	Only needed if you don't want to use the bundled version of APR-Util

So there you have it. When building Subversion from source on Unix you have to be concerned about three things. First, if you want the database back end, you need to be sure you have the correct version of Berkeley DB installed. Second, you need to ensure that your version of APR-Util is linked against that version of Berkeley DB. And third, if you're building `mod_dav_svn`, you need to ensure that you have the correct version of Apache installed and that Subversion can find it.

Of course if you're only concerned with building a Subversion client and you have no desire to create repositories using the database back end, or to run servers that access repositories using the database back end, you don't need to worry about any of this. Just running `configure`, `make`, and `make install` in the Subversion tarball should be enough.

Installing Subversion on Windows Systems

Building Subversion on Windows systems is both considerably more complex than doing so on Unix machines and considerably less common, due primarily to the fact that most Windows users don't have access to the tools necessary to compile software themselves. As a result, virtually everyone who runs Subversion on Windows uses precompiled binaries.

There are two options available to you when it comes to binary distributions of Subversion for Windows. Precompiled versions of Subversion for Windows come as either a ZIP file or a prepackaged installer. The ZIP file simply contains the binaries, the libraries they depend on, and the Apache modules. The installer wraps the same binaries up and automatically installs them for you. You can use whichever you're more comfortable with, although the installer is the easier of the two to use. Unless you know what you're doing it's probably best to simply use the installer, as it minimizes your chances of making a mistake in the installation process.

If you're interested in making modifications to the Subversion source code yourself, or you're trying to debug a problem in Subversion, you can find instructions on how to build Subversion on Windows using Visual C++ 6.x in the `INSTALL` file in the top level of the Subversion distribution. You'll have to obtain most of the same libraries that are used with the Unix build (Neon, and so on); you'll have to build your APR-Iconv in addition to APR and APR-Util; and you'll need both Perl and Python installed for various parts of the process. Needless to say, this isn't for inexperienced developers, and if you decide to just download the precompiled binaries, nobody will think less of you.

If you aren't scared yet, the `INSTALL` file is really the best guide to the process, so we'll just refer you to it from here on.

It's also possible to build Subversion with Visual Studio .NET, but the process is more difficult and not as well documented.

Configuration Files

The client side of the Subversion libraries (and some parts of the server side) make use of a number of configuration files to allow the user to tweak several aspects of the system. Subversion's configuration files make use of an INI-style format, which is compatible with the Python `ConfigParser` module, found at `http://www.python.org/doc/current/lib/module-ConfigParser.html`.

Here's an example of a section from the `~/.subversion/config` file, so you can see what Subversion configuration files look like:

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

In this case, there is a single section, `[auth]`, and within it there is one variable, `store-auth-creds`, which is set to no. Wherever you see a Boolean option, the value can be any of true, false, yes, no, on, off, 1, or 0.

There are two types of configuration files on a system. The first are the systemwide configuration files, which are by default located in `/etc/subversion` on Unix and are optional. The second are the per-user configuration files, which on Unix are located in `~/.subversion`. The first time you run a Subversion command, the `~/.subversion` directory will be created and populated with sample configuration files. Subversion currently makes use of two configuration files: `servers`, which holds options related to connecting to Subversion servers over the network, and `config`, which holds more general options.

On Windows systems, the system configuration files are located in `C:\Documents and Settings\All Users\Application Data\Subversion`,[1] and the per-user configuration files are located in `C:\Documents and Settings\rooneg\Application Data\Subversion`[2] (well, if your username is `rooneg` anyway; if it isn't you'll have to replace that part with your own username). Additionally, you can place configuration options in the Windows Registry instead of in the configuration files. The systemwide configuration is stored in `HKLM\Software\Tigris.org\Subversion`, and the per-user configuration is stored in `HKCU\Software\Tigris.org\Subversion`.

■ **Note** For the rest of the book, we'll be using the Unix-style locations for the configuration files, so if we refer to `~/.subversion/servers`, and you're using Windows, just mentally replace `~/.subversion` with your `Application Data` directory.

For more details on the configuration files and their contents, see the `README.txt` file in the `~/.subversion` directory and the sample configuration files themselves.

Summary

In this chapter, we explained a little about version control in general and Subversion in particular. We covered how to install Subversion on your system, and you learned a little about how Subversion's configuration files work. Now you're ready to move on to Chapter 2, which presents an in-depth tutorial on how to use Subversion.

---

1. OK, technically, they're located in `%ALLUSERSPROFILE%\Application Data\Subversion`, but usually `%ALLUSERSPROFILE%` is `C:\Documents and Settings\All Users`.
2. Again, technically that's `%APPDATA%\Subversion`, but if you're on a typical Windows machine, `%APPDATA%` is the `Application Data` directory in your user profile.