Version Control with Subversion

Matthew Plough
mplough@princeton.edu

November 29, 2006

1 Introduction

Subversion is a version control system, developed as a replacement to CVS, the Concurrent Versions System. Very well written, extensive documentation can be found at `http://svnbook.red-bean.com/`; in this document, I present only a brief overview of the concept of version control and how to use Subversion.

2 Version Control

In COS 333, you have been assigned a group project; hopefully everyone in your group is writing code. Professor Kernighan requires you to use a version control system for a number of reasons. Without one, you'll have problems sharing code; you would need to find a place that all of your group members can access when you aren't all present, and it will be frustrating for all of you to work on the code at the same time.

In the past, you have probably deleted critical files inadvertently, and frantically sought to retrieve them. While trying to find a bug, you have probably added so many changes to a file that you can no longer remember what the code originally looked like, so you may have to rewrite the code completely.

Version control systems address all of these problems, allowing you to write code more efficiently.

Chapter 2 of the Subversion Book, available at `http://svnbook.red-bean.com/`, gives an excellent overview of how version control works. If you have not used a version control system before, it's essential reading, and it's not too long.

2.1 The Repository

All versions of your work are stored in a central location called a repository; any of them can be retrieved at any time. With Subversion, creating a repository and setting up access control are trivially easy; I describe how to do so in Section 3.1.

The repository acts as a "vault" for a project; it changes how you work with your code. Instead of working on the only copy of your project, you check out a copy of the project, make changes, and commit the changes back to the repository. This gives a lot of flexibility; if you make changes that prove useless, you can quickly revert back to a known-good version of your work. Similarly, if you accidentally delete a file, you restore the last version from the repository.

Repositories are generally quite small; only the minimum amount of information needed to construct a given version of a file is stored. If a file does not change between versions, a new copy is not written. If you store your repository on someone's H: drive space, it will be backed up regularly and you will not have to worry about losing your entire project.

2.2 The Working Copy

Once you have finished setting up a repository, any member of your group can check out files from the repository; this local copy of the code is called the working copy. Each group member can make changes to his or her own working copy, and once satisfied with the changes, commit them back into the repository for others to access.

In Subversion, each commit increments a global version number, in contrast with CVS's method of only incrementing the version of changed files. While working with version 648 of your code sounds strange at first, it is much easier to tell someone to check out version 648 than to enumerate versions 1.06, 2.11, 1.55, etc. of N files.

3 Getting Ready to Work

Detailed information on repository administration is available in Chapter 6 of the Subversion Book, at `http://svnbook.red-bean.com/`. This section contains enough information to get you up and running on hats.

3.1 Create a Repository

1. Subversion is installed on hats, so SSH into one of these machines: fedora, fez, or boater.

2. Create a folder called, for example, `svnrepos`; you'll be creating your repository in that folder. You'll want to make it obvious to yourself that the folder contains the repository, not a working copy.

3. Enter that folder by running, for example[1]:

   ```
   $ cd svnrepos/
   ```

4. Create your repository, called, for example, `myrepository`, by running:

   ```
   $ svnadmin create myrepository --fs-type fsfs
   ```

5. Set up access control to your new repository. Run:

   ```
   $ cd myrepository/conf/
   ```

   Now, edit the `svnserve.conf` file to read as follows, replacing `myrepository` with the name you chose:

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

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

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

   Also edit the `userfile` file (if it doesn't exist, create it) to read, for example:

   ```
   [users]
   user1 = password1
   user2 = password2
   ```

   Change the usernames and passwords to the ones that you need – use your netIDs, but don't use your Princeton passwords, since the passwords are being stored in clear text. You won't need the passwords often (your working copy remembers it), but choose something reasonably secure.

3.2 Run a Subversion server

1. Still in SSH, change back to the `svnrepos/` folder.

[1]I use the `$` sign to denote the prompt

2. See if anyone else is currently running Subversion server processes by running this command:

   ```
   $ ps aux |grep svnserve
   ```

   If there are any running, you'll have to use an alternate port number – no big deal. Alternate port numbers are specified using the `--listen-port` argument; check which ports the active servers are using. Pick the first available port number closest to (but greater than) 3690. For example, if servers are using ports 3691, 3692, 3698, and 3700, use port 3693 in step 4.

3. Print the working directory by running:

   ```
   $ pwd
   ```

   Take note of the path; it's probably something like
   `/u/mplough/svnrepos`.

4. Using the path that you got in the last step, run:

   ```
   $ svnserve -d --listen-port [from step 2] -r [path from
   above]
   ```

Copy that command somewhere. Since you ran it as a daemon using the `-d` argument, the `svnserve` program will stay active when you log off, but if OIT reboots the machine you ran it on, you'll have to run it again. Also, take note of the machine you ran the command on and the port that you used – you'll be connecting to that machine on that port in the next section.

Your working copy remembers the server it connected to, so if you have to run the `svnserve` command again, be sure to run it on the same computer.

3.3 Check Out a Working Copy

If you develop under Windows, there is an excellent Windows GUI Subversion client called TortoiseSVN, available at `http://tortoisesvn.tigris.org/`. Subversion has also been ported to Macintosh, BSD, etc., and is available in the package managers of most Linux distributions.

If desired, replace "hats" with the name of your Linux machine or Macintosh, if Subversion is installed on it.

1. SSH into hats.
2. Go into your COS 333 folder or other place you use to store code; the command in the next step will create a folder called, for example, `myrepository`. That folder will be your working copy.

3. In the last section, you ran a Subversion server on one of the machines in the hats cluster – either fedora, fez, or boater. Run the following command using that same machine. Be sure to replace the machine name, the port number, and the repository name with the ones that you have used.

   ```
   $ svn co svn://fedora.princeton.edu:3690/myrepository
   ```

4 Learning to Use Subversion

Read Chapter 3 of the Subversion Book, at `http://svnbook.red-bean.com/`. Before reading, create a test repository accessible by all of your group members. That way, you can try the different commands as you read. Create random files, edit them, and add them. Be sure to have at least two people edit a file, so you can understand how merging works.

4.1 Know how to handle conflicts!

Be especially sure to understand how to handle a conflict. There are two ways, (1) reverting your local changes, and (2) editing the file to taste, and running `svn resolved <filename>`. You can create a conflict with the following procedure:

1. Have two group members check out working copies.
2. Group member A creates a file `foo`, edits it, and adds it to the repository. Group member A commits `foo`.
3. Group member B performs an update.
4. Both group members edit line 5. A changes it to read one thing; B changes it to read something else.
5. A commits.
6. B updates. B's version is now conflicted.

4.2 Use informative commit messages

When committing changes, supply an informative message saying what you did and why. This is useful when other group members want to know what changed, or when you are trying to diagnose the source of a problem. If done correctly, these messages can help you write your documentation.

There are three ways to specify a commit message:

On the command line. When committing, run:

```
$ svn ci [PATH...]  -m 'informative message'
```

This is useful when your message is something trivial like “forgot to add the makefile”.

In a file. You can also write up your changes in a file before the commit, and specify the file containing your message while committing, like so:

```
$ svn ci [PATH...]  -F [file containing your comments]
```

You’ll want to delete the file after writing it, since its contents will be available in the commit log.

With an editor, at commit-time. Recommended. You can also have Subversion open a temporary file in a text editor when you run the commit command. When you save the changes and exit, Subversion will place the file’s contents in the log, and delete the file.

To do this, export the path to your favorite editor in an environment variable called EDITOR in your `.bashrc`, by adding the line (for example):

```
export EDITOR=/usr/bin/vim
```

Then, to commit, run:

```
$ svn ci [PATH...]
```

5 Subversion Commands

For more information on any command, run:

```
$ svn help [command]
```

5.1 Quick Reference

add	Puts files and directories under version control. Usage: `svn add PATH`
co **checkout**	Short for "checkout". Checks out a working copy (see Section 3.3).

ci
commit
	Short for "commit".
	Send changes from your working copy to the repository.
	Usage:
	`svn commit [PATH...]  -m arg`
	`svn commit [PATH...]  -F arg`
	This method is recommended: Define an environment variable `EDITOR=/usr/bin/vim` (for example), and
	`svn commit [PATH...]`

diff	Prints the changes between the last revision and the working copy. Supply a path argument to display changes for just one file.
log	Prints log messages, showing what changes have been made. To see affected paths, use the `-v` argument.
resolved	Resolves the conflicted state of `PATH`. Usage: `svn resolved PATH`
revert	Wipes out local edits, restoring the last revision. This is one way to resolve a conflicted state. Usage: `svn revert PATH`
st **status**	Short for "status". Shows if files are changed, deleted, added, up-to-date, etc. If all files are up-to-date, displays nothing. Supply a `-v` argument to see status of all files.
up **update**	Short for "update". Brings changes in the repository into the working copy.