Cornell University
Center for Advanced Computing

Using the Subversion RCS for Collaborative Research Code Development

Steve Lantz
Senior Research Associate
Cornell CAC

CAC Seminar Series – February 26, 2010

Why Use a Revision Control System (RCS)?

- Good idea to keep all changes to a code in a repository
  - Allows you to undo, backtrack, recover older versions
  - Lets you re-incorporate ideas you previously tried and discarded
  - Provides a convenient way to back up your latest changes
- An RCS provides a sort of “time machine” of a group’s work
  - Keeps logs of the changes that were made, in addition to the code itself
  - Keeps track of who made which changes
- Most RCSs operate across networks
  - Allows them to be used by people on different computers
- RCSs offer a “happy medium” in complexity of use
  - Not as cumbersome as software configuration management (SCM)
  - More sophisticated than rsync, yet it works on any collection of files

Why Use Subversion?

- Subversion is the most popular tool for managing source code
  - More than one-third of software developers use it
  - Its share is 3x that of the next most popular tool, Microsoft SourceSafe
  - Ref.: “Dr. Dobb’s Report,” *Information Week*, Jan. 18, 2010
- Subversion succeeds and improves upon the widely-used CVS
- Subversion repository is centralized: everyone gets the latest
  - Drawback: need to have access to the centralized server to update
  - Distributed RCSs (Git, Mercurial) are gaining ground over client-server
- Cornell will host your Subversion repository at forge.cornell.edu
  - The Cornell SourceForge site provides many other collaboration tools, including ones for bug tracking, task planning, document management, wiki documentation, discussion forum, etc. – all backed up regularly

Typical Work Cycle

- Get or update a working copy
  - svn checkout
  - svn update
- Make changes
  - svn add
  - svn delete
  - svn copy
  - svn move
  - svn mkdir
- Possibly undo some changes
  - svn revert
- Examine your changes
  - svn status
  - svn diff
- Resolve conflicts (*merge others' changes*)
  - svn update
  - svn resolve
- Commit your changes
  - svn commit

Do an Initial Checkout

- Checking out a repository creates a “working copy” of it on your local machine. This copy contains the HEAD (latest revision):

```
$ svn checkout http://svn.collab.net/repos/svn/trunk
A    trunk/Makefile.in
A    trunk/ac-helpers
A    trunk/ac-helpers/install.sh
A    trunk/ac-helpers/install-sh
A    trunk/build.conf
Checked out revision 8810.
```

- To place a working copy in new directory *subv* instead of *trunk*:

```
$ svn checkout http://svn.collab.net/repos/svn/trunk subv
```

Why “Checkout” is a Misnomer

- Subversion uses *copy-modify-merge* instead of *lock-modify-unlock*.
  - Your working copy is just like any other collection of files and directories on your system.
  - You can edit and change it, move it around, even delete the entire working copy and forget about it.
  - There's no need to further notify the Subversion server that you've done anything; no “check-in” is required before others can work.
- But: don’t delete or change anything in the .svn directories!
  - Subversion depends on these to manage your working copy.
- You can just as easily check out any deep subdirectory of a repository by specifying its path in the checkout URL.

Sync With the Latest Revision

```
$ svn update
U  foo.c
U  bar.c
Updated to revision 2.
```

- It appears that someone checked in modifications to both foo.c and bar.c since the last time you updated. Subversion has updated your working copy to include those changes.
- To learn what the letter codes mean, run *svn help update*.
- We'll be returning to *svn update* later.

Modify Your Working Copy

- Use *svn add*, *svn delete*, *svn copy*, *svn move*, and *svn mkdir* when making “tree changes” to the directory structure.
- These commands schedule operations that will actually take place in the repository the next time you commit.
- If you are merely editing files that are already in Subversion, you may not need to use any of these commands.
- You can give a URL instead of a filename to these commands; it’s the same as doing an immediate commit (not advisable).

More on Tree Changes

`svn add foo; svn mkdir bar`

- If foo is a directory, the whole tree will be added unless this is prevented with *--depth empty*
- The latter is the same as: *mkdir bar; svn add bar*

`svn delete foo`

- Deletes foo (not from the repository, just from its next head)

`svn copy foo bar; svn move foo bar`

- Like: *cp foo bar; svn add bar* – but records source of copy
- The latter is the same as: *mv foo bar; svn delete bar*

See an Overview of Your Changes

- Run *svn status* at the top of your working copy, no arguments

```
?    scratch.c           # file is not under version control
A    stuff/loot/bloo.h   # file is scheduled for addition
C    stuff/loot/lump.c   # file has textual conflicts from an update
D    stuff/fish.c        # file is scheduled for deletion
M    bar.c               # the content in bar.c has local modifications
```

- Conflict explanation
  - Changes received from the server during an update overlap with local changes that you have in your working copy (and resolution was postponed at the time of the update).
  - You must resolve this conflict before committing your changes to the repository.

More Ways to Use *svn status*

- If you pass a specific path to *svn status*, you get information about that item alone.
- The *--verbose* (*-v*) option will show you the status of every item in your working copy, even if it has not been changed.
- Most invocations of *svn status* do not contact the repository; they compare the metadata in .svn with the working copy.
- However, the *--show-updates* (*-u*) option contacts the repository and adds information about things that are out of date.

Examine Local Modifications

- The *svn diff* command compares your working files against the cached “pristine” copies within the .svn area.
- Output is displayed in unified diff format; that is, removed lines are prefaced with -, and added lines are prefaced with +.
- Files scheduled for addition are displayed as all added text, and files scheduled for deletion are displayed as all deleted text.
- You can generate “patches” by redirecting the diff output to a file.

Update (and Deal with Conflicts)

- We've seen that *svn status -u* can predict conflicts. Suppose you run *svn update* and find the repository has changed:

```
$ svn update
U  INSTALL
G  README
Conflict discovered in 'bar.c'.
Select: (p) postpone, (df) diff-full, (e) edit,
        (h) help for more options:
```

  - Files marked with U contained no local changes but were Updated with changes from the repository.
  - The G stands for merGed, which means that the changes coming from the repository didn't overlap with local ones.

Ways to Resolve Conflicts

- Here's what you see when you ask for more options with "h":

```
(p)  postpone     - mark the conflict to be resolved later
(df) diff-full    - show all changes made to merged file
(e)  edit         - change merged file in an editor
                    [use $EDITOR]
(r)  resolved     - accept merged version of file
(mf) mine-full    - accept my version of entire file
                    [ignore their changes]
(tf) theirs-full  - accept their version of entire file
                    [lose my changes]
(l)  launch       - launch external tool to resolve conflict
(h)  help         - show this list
```

Postpone Conflict Resolution

- You make changes to the file *sandwich.txt*, but you do not yet commit those changes. Meanwhile, Sally commits changes to that same file. You update your working copy before committing and you get a conflict, which you postpone (p):

```
$ svn update
Conflict discovered in 'sandwich.txt'.
Select: (p) postpone, (df) diff-full, (e) edit,
        (h)elp for more options : p
C  sandwich.txt
Updated to revision 2.
$ ls
sandwich.txt       sandwich.txt.r1
sandwich.txt.mine  sandwich.txt.r2
```

Resolve Conflicts

- If you've postponed a conflict, you will need to resolve it before Subversion will allow you to commit your changes. You can do this using *svn resolve --accept,* plus an argument:
  - *base* chooses the version of the file that you last checked out before making your edits.
  - *mine-full* chooses the version with only your edits.
  - *theirs-full* chooses the version that your most recent update retrieved from the server (discarding your edits entirely).
  - If you'd prefer to pick and choose between your changes and the changes that *svn update* fetched from the server, you can merge the conflicted text "by hand" (by examining and editing the conflict markers within the file), then choose the *working* argument.

Resolve Conflicts by Hand

```
$ cat sandwich.txt
Top piece of bread
Mayonnaise
Lettuce
Tomato
Provolone
<<<<<<< .mine
Salami
Mortadella
Prosciutto
=======
Sauerkraut
Grilled Chicken
>>>>>>> .r2
Creole Mustard
Bottom piece of bread
```

- Usually you won't want to just delete the conflict markers and Sally's changes—she's going to be awfully surprised when the sandwich arrives and it's not what she wanted.
  - This is where you pick up the phone or walk across the office and explain to Sally that you can't get sauerkraut from an Italian deli.
  - Once you've agreed on the changes you will commit, edit your file and remove the conflict markers (lines with <,=,>).

Outcome of Resolution

- *svn resolve* removes the three temporary files and accepts the version of the file that you specified with the *--accept* option. (You must explicitly list the filenames you wish to resolve.)
- Use it only when you're certain that you've fixed the conflict in the file. Once the temporary files are removed, Subversion will let you commit the file even if it still contains conflict markers.
- At this point, Subversion no longer considers the file to be in a state of conflict and you can commit it to the repository.

```
$ svn resolve --accept working sandwich.txt
Resolved conflicted state of 'sandwich.txt'
$ svn commit -m "We'll go with my sandwich, discarding Sally's edits."
```

Commit Your Changes

- When you commit a change, you must supply a log message.
  - If the message is brief, you may wish to supply it on the command line using the *--message* (or *-m*) option:

```
$ svn commit -m "Corrected number of cheese slices."
Sending        sandwich.txt
Transmitting file data .
Committed revision 3.
```

  - If you've been composing one as you work, you may want to pass the filename to Subversion with the *--file* (*-F*) option.
  - If you fail to specify either *-m* or *-F*, Subversion will automatically launch your favorite editor for composing a log message.

Repeat the Update if Necessary

- If someone has changed any of the files involved in your commit, the entire commit will fail with a message informing you that one or more of your files are out of date:

```
$ svn commit -m "Add another rule"
Sending        rules.txt
svn: Commit failed (details follow):
svn: File '/sandwich.txt' is out of date
...
```

- At this point, you need to re-run *svn update*, deal with any merges or conflicts that result, and attempt your commit again.

Recover from an Interruption

- Before updating the working copy, Subversion writes its intentions to a logfile, just as in a journaled file system.
  - If a Subversion operation is interrupted (e.g., if the process is killed or if the machine crashes), the logfiles remain on disk.
  - By reexecuting the logfiles, Subversion can complete its operation, so your working copy can get itself back into a consistent state.

- *svn cleanup* searches your working copy and runs any leftover logs it finds, removing any locks in the process.
  - If Subversion ever tells you that some part of your working copy is "locked," this is the command that you should run.
  - *svn status* will display an L next to locked items.

Other Basic *svn* Commands

- *svn import* is a quick way to copy an unversioned tree of files into a repository (follow it with *svn checkout* to begin work).

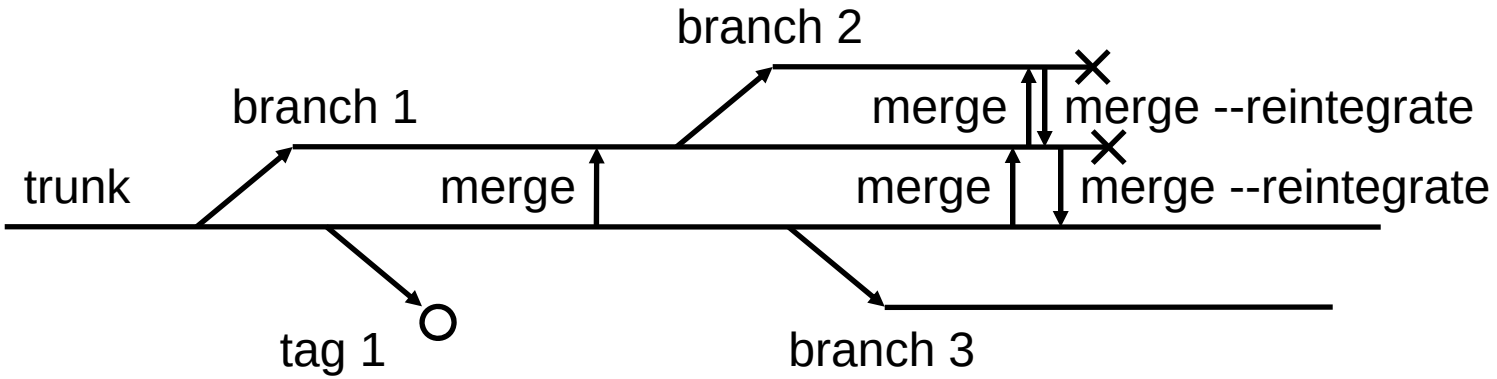
```
$ svn import mytree file:///var/svn/repos/some/prj -m "Initial import"
```

- *svn log* shows log messages with the date and author attached to each revision, and the paths that changed in each revision.

- *svn list, svn cat* display directories or files for a given revision.

- *svn export* is like a checkout, except it omits .svn information.

Branching, Merging, and Tagging



- Thus far we've assumed all collaborators are committing their revisions to a single "trunk" version of the code.
- However, in a Subversion repository, one can also create distinct *branch* or *tag* versions, which begin simply as a copies of the trunk.
  - A tag is a branch with no further development.
  - Designations like trunk, branch, and tag are mostly conventions.

Create a Branch

- Let's say it's our policy to create branches in the /calc/branches area of the repository, and you'd like to name your new branch my-calc-branch. It should begin life as a copy of /calc/trunk.

```
$ svn copy http://svn.example.com/repos/calc/trunk \
        http://svn.example.com/repos/calc/branches/my-calc-branch \
        -m "Creating a private branch of /calc/trunk."
```

- Note, *svn copy* can be used to do a "remote" copy entirely within the repository. You just copy one URL to another.
  - Yes, this causes a near-instantaneous commit in the repository.
  - But it's *much* faster than doing an *svn copy* in the working copy.

Get Your Branch Up to Date

- After some time, you may want to sync your branch with the trunk, to make sure you have the latest revisions to the trunk.

```
$ pwd
/home/user/my-calc-branch
$ svn merge http://svn.example.com/repos/calc/trunk
--- Merging r345 through r356 into '.':
U    button.c
U    integer.c
```

- As with *svn update*, you may need to resolve some conflicts.
- If testing looks good, you can commit your up-to-date branch.

Keep Your Branch Up to Date

- At some future time, you may want to sync your branch again.

```
$ svn merge http://svn.example.com/repos/calc/trunk
--- Merging r357 through r380 into '.':
U    integer.c
U    Makefile
A    README
```

- Subversion remembers which revision you last merged with: it stores this information in your metadata (.svn directories).
- You can use *svn status* and *svn diff* to examine the specific changes caused by each sync with the trunk.

Merge Back into the Trunk

- First, do a final commit of your branch.
- Change directories to your clean, up-to-date working copy of the trunk (obtained with *svn checkout*, say). Then:

```
svn merge --reintegrate \
   http://svn.example.com/repos/calc/branches/my-calc-branch
--- Merging differences between repository URLs into '.':
U    button.c
U    integer.c
U    Makefile
U    .
... [Build, test, verify, etc.] ...
$ svn commit -m "Merged my-calc-branch back into the trunk
```

Reference

Version Control with Subversion, 2nd Edition

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

Publisher: O'Reilly Media, Inc.

Pub Date: September 23, 2008

Print ISBN-13: 978-0-596-51033-6

Pages: 432

http://proquest.safaribooksonline.com/9780596510336

(free access from any Cornell IP address)