RSS webmaster

Reference Guide to Subversion (SVN) Source Control System for use in RSS

Subversion is a powerful, open-source version control system favored by the RSS course staff for use by RSS teams doing shared code development. This guide is a primer to the use of Subversion within RSS. Subversion provides a simple command set with which you can:

- Check out a local, working copy of code from a shared repository;
- Track the status of your local copy with respect to the repository;
- Commit your local changes back into the repository;
- Update your local copy to reflect changes committed by others;
- Merge sources and resolve conflicts as needed; and
- Revert to an earlier, stable version if anything goes awry.

The on-line version of this document (posted in the Labs/ area of the RSS home page) links to documentation on the Subversion web site. Many of the example Subversion command lines shown below are derived from that site.

Getting Started

As alluded to in lecture and lab, Subversion supports a source control model in which multiple users share read/write access to a source *repository*, which represents the group's consensus version of its collective code-base or *source tree*. Users modify the repository *only* through a well-defined set of actions implemented by the program `svn`, which "manages" files on the user's behalf. A user wishing to add or modify code in the repository must first retrieve or "check out" a local working copy of the repository. S/he then edits the local version and tests it locally. When the user's local copy is once again functional (and hopefully, improved), the user can "commit" his/her changes to the repository, at which time the modified code becomes visible to other users. All users can monitor the status of the repository, and the state of their own local files with respect to their (presumably stable) counterparts there. Finally, users can "update" and "merge" their local copies to incorporate changes committed by others to the repository, "resolve" conflicts between their own edits and changes made by others, and "revert" to earlier committed versions if necessary.

Checking Out Your Group Repository

First, you may wish to add a line like this to your shell's run-time command file (e.g. ~/.bashrc) to reflect your choice of editor (this just controls which editor gets invoked when svn needs you to edit a file or enter a commit message, as described below; your account in the hangar will use emacs here by default):

```
export SVN_EDITOR=vim # or other text editor of your choice
```

If you do so, then update your active shell to reflect these changes (either by cutting and pasting the lines above to the shell and executing them, or by exiting your shell and starting a new one).

Decide where in your local space (e.g. "rss") you will store your RSS files, and make that directory if you haven't already:

```
mkdir rss
```

Now cd into your working area:

```
cd rss
```

And check out a copy of your group's source tree, where *N* is your one-digit group number and *user* is your rss username

```
svn checkout file:///home/group-N/SVNROOT/RSS-I-group
```

(use your rss password if prompted).

Editing and Committing Changes

Change directories down into your local source tree. Now you can begin collaborative code development with your teammates. Each of you should use the editor of your choice. We've provided one display, keyboard and mouse per workstation, but if you bring a laptop into the lab, you can `ssh` in and work simultaneously with your teammates.

Once you have edited, compiled, tested, and documented one or more files, you will use "svn commit" to propagate your local revisions back into the repository, incrementing the version number of the repository file as a side-effect. You can either do this on a per-file basis (recommended for beginners):

```
svn commit filename
```

Or you can simply rely on Subversion to figure out exactly what has changed in the sub-tree rooted at the current directory, and commit it:

```
svn commit
```

For each commit action, Subversion will bring up an editor and ask you to enter a brief comment describing the purpose of the commit. These comments will be displayed to you and your teammates when you ask Subversion for the edit history of a file, and are among the most valuable metadata (along with filenames and modification dates) that Subversion associates with your code. Even if you feel that the changes you just made are completely obvious in their purpose and function, they may not be obvious to your teammates or the staff, and frankly they may not be obvious to you either after a few hours have passed. So, **type in a brief, meaningful comment whenever Subversion asks you to**.

Typically the commit will occur successfully. But if someone else has committed changes since your most recent checkout or update, you'll have to merge their changes with yours, resolving conflicts as needed before Subversion allows the commit (see below).

Adding and Removing Files

Suppose you decide to design a new Java class, and create a file `MyClass.java` to implement it. Subversion's data model, in which each user edits only a local copy of the source tree, has some implications that take a bit of getting used to. For example, there's no such thing as adding a file directly to the repository. Instead, the user creates the file locally (typically with a text editor), then adds it to the local tree, then commits it to the repository:

```
emacs MyClass.java    # create local copy
svn add MyClass.java  # schedule file for addition to repository
svn commit            # propagate file MyClass.java to repository
```

Similarly, to delete a file, a user must first delete the local copy of the file, then inform Subversion that it should be deleted from the repository as well (otherwise, Subversion will simply replace the file as part of the next update):

```
rm MyClass.java           # delete local copy
svn delete MyClass.java   # schedule file for removal from repository
                          # MyClass.java must not exist in current directory
svn commit                # propagate deletion of MyClass.java to repository
```

Use `svn add` and `svn delete` wisely. Don't add temporary files (editor backup files, .o files from compilation, etc.). Don't

delete files added by another user without first talking to that user. You can use `svn revert` (see below) to correct these sorts of errors before they are committed.

Checking Status, and Updating to Incorporate Changes Made by Others

In a source tree with dozens or hundreds of files shared by even a few people, it will naturally occur that two or more people change the same file at about the same time. This happens even when teammates try hard to keep each other informed about the edits they plan to make. To facilitate communication about overlapping edits, Subversion provides a mechanism called "svn status" that compares a local file or subtree to its counterpart in the repository. As with commit, status can be requested for a single file:

```
svn status filename
```

Or, status can be requested for the sub-tree rooted at the user's current directory:

```
svn status
```

In either case, `svn` produces a listing of every file that differs from its repository counterpart, and for each file a one-letter code describing the difference. There are many one-letter codes, but the most important for our purposes are:

```
M foo       (user's copy of foo has been locally modified)
C foo       (file has conflicts from an update)
A foo       (file is scheduled for addition to repository)
D foo       (file is scheduled for deletion from repository)
L  foo       (svn has "locked" foo; at this moment, it is managing foo for another
? foo.o     (foo.o is not part of the repository; svn does not manage it)
```

Locked files can be a mysterious part of version control. Locks usually clear after a few seconds as the (other user's) locking operation completes. Occasionally, though, locks persist indefinitely. This can happen due to network disruptions or (more likely) Subversion terminating prematurely on the client or server due to an error condition of some kind. If you see a lock that lasts more than a minute, call an RSS staff member over or email `rss-help` for assistance.

Merging, and Resolving Conflicts with, Changes Made by Others

Typically, team members communicate in order to avoid simultaneous revision of the same file. However, in some cases (for example when two people add methods to the same class), overlapping changes are hard to avoid. This is not a problem for the first person to commit her changes; Subversion simply accepts the commit with no errors or warnings, as usual. However, the second person to attempt a commit will find that her local copy is not up to date with the repository version, and will be required to merge any conflicts before committing. Subversion provides several tools to support file merging:

```
svn log filename              # shows history of file over time
svn status filename          # shows current file status, see above
svn status –u filename       # show status, and predict conflicts on update
svn diff filename            # diffs file with its committed counterpart
```

It is good practice to invoke `svn status` frequently, especially before committing your own changes, to see what conflicts (if any) your local changes may cause with material already committed to the repository. Again, **if others have modified overlapping areas of the repository, Subversion requires that you incorporate or "merge" their changes** into your local tree before committing your own changes:

```
svn update
```

The update command will print a one-letter code for each updated file: "`U`" (file updated from repository), "`G`" (file merged with non-overlapping updates from repository), or "`C`" (conflict found between your local edits and the repository version of the file, typically because someone else has committed changes in the same region of the same file).

Subversion inserts marker lines (typically <<<<<, =====, and >>>>>) into conflicting files to demarcate conflict regions. You will have to inspect and reconcile these regions in your editor. Subversion tries to help you by placing files with the same base name as the conflicted file, but suffixes "`.mine`," "`.rNEWREV`", and "`.rOLDREV`", in your working directory, which represent your pre-update local copy, the repository's pre-checkout committed copy, and the repository's current committed copy, respectively. Note that you'll almost always want to remove the conflict demarcation lines as part of your resolution process, since the presence of such lines in program text (as opposed to within comments) will prevent compilation. Once you have resolved the conflicts to your satisfaction, remove these temporary files using `rm` and invoke:

```
svn resolved filename
```

to inform Subversion that you have resolved the conflict. Subversion will now allow the file to be committed.

For advanced Subversion users, "svn merge" can patch specified committed versions into a working version.

Reverting to an Earlier Version

Sometimes, you'll realize that you've made a mistake: pursued an ill-advised implementation strategy, accidentally edited the wrong file, added a temporary file, or deleted source files that you actually cared about, for example. In fact, Subversion already provides a safety net for such cases, called "`svn revert`". This command requires at least one argument: either a list of one or more filenames, or a `--recursive` flag and location of an entire directory to revert:

```
svn revert filename          # revert a single file
svn revert --recursive .     # revert entire working directory
```

The `revert` action is comprehensive; in addition to undoing any local edits you have made, it "unschedules" any file additions or deletions you have scheduled (but not yet committed).

You can also revert to earlier versions across commit boundaries, using "`svn update --revision <version>`", where the version specifier can be an absolute version number, a relative version number, or a date (or date/time) specifier:

```
svn update --revision PREV foo                # decrement foo's working revision
svn update --revision 1729                    # updates existing working copy to r

svn update --revision {2005-02-17}            # update to 12:00:00am, start of Feb
svn update --revision {15:30}                 # update to 2:30pm today
svn update --revision {"2005-02-17 15:30"}    # update to 2:30pm on February 17th
```

If you would rather not modify your working copy, create a directory elsewhere and use the "`svn checkout`" form of the commands above to check out fresh local copy to that directory.

Be very careful with revert and update: they can wipe out hours of hard work in an instant. Remember that revert and update actions may have irreversible effects on your (previous) working copies.

Advanced Subversion Capabilities

Subversion is a distributed file-sharing and file-versioning system with extensive local caching. Its actions often involve the network, take a significant amount of time to complete, and can be interrupted or otherwise fail. For these (hopefully infrequent) failure cases, Subversion provides a "cleanup" command that uses logs to put your working copy back into a consistent state. (Take 6.033 for more about logs and transaction consistency.)

Subversion has many other powerful capabilities. It can be configured to email all team members whenever any team member commits a change to the repository. It supports branches and tags, general metadata association with files, and a host of other complex functions whose descriptions are beyond the scope of this document. Refer to the on-line reference "Version Control with Subversion" (Collins-Sussman, Fitzpatrick, and Pilato) for comprehensive documentation.

General Tips

This section collects a few words of wisdom gained through the RSS staff's experience with a variety of source control systems. These are meant to be read with care, along with the admonitions mixed in to the descriptions of Subversion functionality above. We hope that you'll heed these tips in your own software development. But you'll probably learn these lessons the hard way, like we did.

Version control systems are meant to enhance teamwork and communication, not replace it. Don't stop talking to your teammates just because you're using Subversion. If anything, the sharing model imposed by the system should prompt you to talk (and think!) more with your teammates, about exactly who is going to make what changes to your source, when, and how. More talk equals less stepping on each others' toes with unintended changes and consequent unpleasant merges and conflicts. **Communicate with your teammates.**

Ignore the "`.svn`" subdirectory and its contents. Every directory checked out with `svn` contains one, which Subversion uses to hold local versioning information and a partial cache of the repository. **Do not modify or delete anything in this directory**. (If you do, it will end in tears.)

Check status and update frequently, especially for files you are about to edit. Checking status tells you when your local copy is out of date, i.e. when someone else has committed changes to the repository since your most recent checkout or update. Doing an update brings you up to date by incorporating those changes into your local copy. There's no sense charging ahead to edit anything until you've got your teammates' most recent versions. If you do, you're just setting yourself up for a possibly painful merge and/or conflict task later, whenever you decide to commit your own changes.

Exit your editor buffers when updating, at least until you become completely comfortable with your editor and Subversion's repository model. Otherwise you can get into a quite confusing situation in which your editor overwrites new, good code from your teammates with your own older, not-as-good code. Some editors even do this automatically (example: "emacs auto-save"), thinking they are doing you a favor. (They're not.)

Commit stable versions often – typically whenever you make a substantive functional change. Each commit provides a potential "island of stability," amongst a sea of changes, to which you and your teammates can revert in case of problems. Remember, if you work for eight hours and commit only once at the end, then the only reversion point available to you will be to the state you were in before all of your work was started. Don't be afraid to commit often. Don't worry; the `svn` tools can handle it. (But be reasonable; committing every time you add a comment or some whitespace to your code is going too far.)
