Version control for Linux

An overview of architectures, models, and examples

M. Tim Jones

October 16, 2006
(First published October 10, 2006)

Version control systems, or source management systems, are an important aspect of modern software development. Not using one is like driving a car too fast: it’s fun and you might get to your destination faster, but an accident is inevitable. This article provides an overview of Software Configuration Management (SCM) systems and their benefits, including CVS, Subversion, Arch, and Git. It also reviews the most common SCM architectures. Finally, it explores some of the new approaches that are available and how they differ from the earlier methods. [Listing 4 has been updated to reflect improvements to Git’s syntax. -Ed.]

What is Software Configuration Management?

SCM is one of the most important tools you probably didn’t learn in school. Software (or source) control, as the name implies, is a tool and an associated process that is used to maintain source code and its evolution over time. SCM provides these primary capabilities:

- Maintains files in a repository
- Maintains revisions of files in a repository
- Detects source change conflicts and provides merging for multi-developer environments
- Tracks originators of changes
- Provides configuration management of files (relation of revisions) for consistent and repeatable builds

**Applicability of SCMs**

Source control typically implies the control of source code and associated files, whereas source management can apply to any type of asset. A Web site consisting of Hypertext Markup Language (HTML) and binary image files, general text documents, or any other file is a candidate for revision control by an SCM system.

So, an SCM allows you to control a set of files in a repository and track revisions of those files. When changes are made to files in the repository by a different developer, the SCM can identify
conflicts from your changes and either automatically merge them or notify you of the conflict. This is an important capability because it permits multiple developers to modify the same set of files. An SCM also provides accountability in tracking who made which changes. Finally, an SCM allows you to logically group files together into sets that are related, such as source files that make up a software image or executable.

The language of SCM

Before you dive too deeply into the details and types of architectures for SCMs, you need to learn the vocabulary. First, there's a repository. The repository is a central location where files are stored and managed (sometimes referred to as a tree). Getting files from the repository to the working folder of your local system is called a check-out. If you make changes to your local files and you want to sync up with changes at the repository, you perform an update. To check your changed files back into the repository, you perform a commit. If your changed file was previously changed and committed by someone else, a merge occurs, meaning the two changesets are brought together. When a merge can't take place because of conflicting changes to a file, a conflict has occurred. In this situation, the commit is rejected, requiring the developer to merge the changes by hand. When a change is committed, a new revision is created for the file.

It's possible for one or more developers to operate off of the main tree (the current head of the repository) or a personal branch that sits on the side of the main tree. This allows developers to try things on their branches without affecting the main tree. When they are stable, you can merge branches back to the main tree.

To mark an epoch in the evolution of a source tree, you can apply a tag to a set of file revisions. This groups the set of files together as a useful collection (sometimes used as a release of the files for a unique build).

Architectures

SCMs can differ in significant ways, but there are two fundamental architectural differences that are worth exploring:

- Centralized versus distributed repositories
- Changeset versus snapshot models

Centralized vs. distributed repositories

One of the most important architectural differences in modern SCMs that you can see and feel is the idea of a centralized versus a distributed (or decentralized) repository. The most common architecture found today is the centralized repository. This star architecture is illustrated as a central source repository with multiple developers working around it (see Figure 1). Developers check out source code from the central repository into a local sandbox and, after making changes, commit it back to the central repository. This allows other developers to access their changes.
Figure 1. In a centralized architecture, all developers work from a central repository

Branches can also be created at the central repository, allowing multiple developers to collaborate on a set of changes to the source at the repository, but outside of the mainline (or tip).

The distributed architecture allows developers to create their own local repositories for their changes. The local developer repository is similar to the original source repository (it's been distributed). The key difference is that instead of sandboxes, where changes are made in the centralized approach, the distributed approach allows developers to work with their repositories while disconnected. They can make changes, commit them to their local repositories, and merge changes from others without affecting the main branch. Developers can then make changesets available to upline developers (see Figure 2).
The decentralized architecture is interesting because it allows independent developers to work asynchronously in peer-to-peer networks. When work is ready (and preferably stable), they can distribute changesets (or patches) to make features available to others. This is the model for many open source systems today, including the Linux® kernel.

**Snapshot vs. changeset models**

Another interesting architectural difference between older and more recent SCMs is how delta changes are stored. They are theoretically the same and yield the same results, but they differ in how revisions are stored.

In the snapshot model, complete files are stored for the entire repository for each revision (with optimizations to reduce the size of the tree). In the changeset model, only the deltas are stored between revisions, creating a compact repository (see Figure 3).
Figure 3. The snapshot and changeset models each offer unique advantages

As you can see in Figure 3, the models differ but have the same result. In the snapshot model, you can get revisions quickly, but you need more space to store them. The changeset model requires less space, but it may take more time to get a particular revision because a delta must be applied to the base revision. As you'll see later, you can make optimizations to minimize the number of deltas that must be applied.

Example SCMs

Let's look at a number of SCMs split out by their architecture: centralized versus distributed. As you'll see shortly, some SCMs can even support both models.

CVS

Concurrent Versions System (CVS) is one of the most common SCMs around today. It's a centralized solution using the changeset model in which developers work with a centralized repository to collaborate on software development. CVS is ubiquitous and is available as a standard part of any Linux distribution. Its simple and comfortable (to many of us) syntax makes it a common choice as a multi- or single-developer SCM.

Listing 1 shows a sample set of CVS commands along with short descriptions of each. For more CVS information, see the Related topics section.
Listing 1. Sample commands for CVS

# Create a new repository
cvs -d /home/user/new_repository init

# Connect to the central repository
echo CVSROOT=:pserver:user@example.com:/cvs_root

# Check out a sandbox for module project from the central repository
cvs checkout project

# Update a local sandbox from the central repository
cvs update

# Check in changes from the local sandbox to the central repository
cvs commit

# Add new files to the local sandbox (need to be committed)
cvs add <file/subdirectory>

# Show changes made in the local sandbox

cvs diff

For you point-and-clickers out there, CVS has a number of open source graphical front-ends that you can use, including WinCVS and TortoiseCVS (which integrates with Microsoft® Windows Explorer, if you enjoy that).

While CVS enjoys wide adoption, it has its warts. CVS doesn't allow you to rename files, and it doesn't work well with special files, such as symlinks. Changes are tracked by file instead of per change, which can be annoying. Merges can sometimes be problematic (CVS internally uses diff3 for this purpose).

However, CVS is useful, does what it needs to do, and is available for all major platforms. If you like CVS, but not its issues, then Subversion may be what you're looking for.

Subversion

Subversion (SVN) was designed as a direct replacement for CVS, but without its previously defined issues. Like CVS, Subversion is a centralized solution and uses the snapshot model. Its commands mimic those of CVS but with a few additions to handle things such as removing files, renaming files, or reverting to the original file.

Subversion also permits remote access via a number of protocols, such as Hypertext Transfer Protocol (HTTP), secure HTTP, or the custom SVN protocol that also supports tunneling through Secure Shell (SSH).

Listing 2 explores some of the commands supported in Subversion. I've also included some of the extensions that aren't available in CVS. See the Related topics section for more information about Subversion. As you see, Subversion's command set is similar to CVS's, making it a great alternative for CVS users.

Listing 2. Sample commands for Subversion

# Create a new repository
svnadmin create /home/user/new_repository
# Check out a sandbox from the central repository
svn checkout file:///server/svn/existing_repository new_repository

# Update a local sandbox from the central repository
svn update

# Check in changes from the local sandbox to the central repository
svn commit

# Add new files to the local sandbox (need to be committed)
svn add <file/subdirectory>

# Show changes made in the local sandbox
svn diff

# Rename a file in the local sandbox (requires commit to the repository)
svn rename <old_file> <new_file>

# Remove files (also removed from repository, requires commit)
svn delete <file/subdirectory>

Following CVS, Subversion integrates into graphical front-ends such as ViewCVS and TortoiseSVN. Tools also exist to convert a CVS repository to Subversion (such as cvs2svn.py), but they reportedly don't handle all branching and tagging cases of complex repositories. As with all open source projects, time will improve this. Subversion also integrates TortoiseMerge as a difference viewer and patch program.

Subversion fixes a number of issues suffered by CVS users, such as versioning of special files and atomic commits and checkouts. If you like CVS and you're committed to the central repository approach, then Subversion is the SCM for you.

Now let's depart from the centralized approach and step into what some believe is the real future of SCM: collaborative decentralized repositories.

**Arch**

Arch is a specification for a decentralized SCM that offers many different implementations. These include ArX, Bazaar, GNU arch, and Larch. Arch not only operates as a decentralized SCM (as shown in Figure 2), but also uses the changeset model (see Figure 3). The Arch SCM is a popular method for open source development because developers can develop on separate repositories with full source control. This is because the distributed repositories are actual repositories complete with revision control. You can create a patch from changes in the local repository to be used by an upstream developer. This is the real power of the decentralized model.

Like Subversion, Arch corrects a number of issues found in CVS. These include metadata changes such as revisioning file permissions, handling file deletion and renaming, and atomic checkins (grouping checkins together instead of as individual files).

Listing 3 shows some of the commands that you find in an Arch SCM. I've chosen to demonstrate GNU arch here because it's developed by the Arch architect, Tom Lord. GNU arch provides the basics you expect from an SCM, including the newer features found in Subversion.
Listing 3. Sample commands for GNU arch (tla)

```bash
# Register a public archive
tla register-archive http://www.mtjones.com/arch

# Check out a local repository from the upstream repository
tla get project@mtjones.com-dev/project--stable myproject

# Update from the local repository
tla update

# Check in changes to the local repository
tla commit

# Add new files to the local repository (need to be committed)
tla add <file>

# Show changes made in the local repository (patch format)
tla what-changed

# Rename a file in the local repository (requires commit to the repository)
tla mv <old_file> <new_file>

# Remove files (also removed from repository, requires commit)
tla rm <file>
```

Arch also allows merging of changes from upstream repositories with `star-merge`. To minimize the number of patches that must be applied to a base revision (per the changeset model), the `cacherev` command will create a new snapshot of the base revision in the repository.

An advantage to Arch is that while it was designed for decentralized operation, it can also be used in the centralized repository paradigm.

The biggest complaint from new users of `tla` is that it tends to be a little complicated. Other implementations of Arch, such as `baz`, are reportedly simpler. You can explore them if `tla` doesn't meet your needs.

Now let's look at one final decentralized SCM written by the maintainer of the Linux kernel himself, Linus Torvalds.

**Git**

The Git SCM was developed by Linus Torvalds as a replacement for the Bitkeeper SCM (see the Related topics section). It's very simple, but it does the job of a decentralized changeset-based SCM and is used as the SCM for the Linux kernel. It uses a file-group model rather than tracking single files. The changesets are compressed and hashed with SHA1 to verify their integrity (see Listing 4).

**Listing 4. Sample commands for Git**

```bash
# Get a Git repository (first time)
git clone \
git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux-2.6.git

# Update a Git repository from the defined upstream Git repository
git pull
```
The Git SCM is self-hosted in its own Git repository, which means that you must bootstrap Git to install it on your local machine. The command set for Git is similar to what you've seen thus far, but it's relatively basic.

You might well ask, why not use one of the existing SCMs that are out there? That's a good question. Git is interesting and serves a large user base of Linux kernel hackers, so it could be the next big SCM. Linus describes Git as a very fast directory content manager that doesn't do much, but does it efficiently.

**Benefits**

Whichever type of SCM you use, there's a universal set of benefits that you reap. With an SCM, you can track changes to files to know how your software has evolved. When incorrect changes are made, you can find them and revert them to the original source. You can group sets of file revisions together and tag them to make releases that can be checked out at any time to repeatedly build specific releases of code (a requirement of SCM).

Whether you use a centralized or distributed repository, snapshot or changeset model, the benefits are the same. Since no modern software development project can be without an SCM, use them early and use them often!

**Looking further**

This article as must scratched the surface of SCMs in use today. Many other open source SCMs exist, including Aegis, Bazaar-NG, DARCS, and Monotone, to name a few. Like editors and languages, SCMs tend to result in strong debates with no correct answer. If you're productive with a tool, use it! SCMs can be problematic because they're rarely used in isolation and, therefore, are usually chosen by teams rather than individuals (unless you have an autocratic boss who likes to make decisions for you). Therefore, play with the possibilities and become comfortable with a few different styles. SCM is a necessary tool in software development and a valuable part of your engineering toolbox.
Related topics

- Explore a large number of SCMs for Linux at LinuxMafia.
- Read David Wheeler's interesting paper on Open Source SCMs, covering CVS, Subversion, Arch, and Monotone.
- CVS is one of the oldest and most widely used SCMs.
- Subversion is a compelling alternative to CVS.
- GNU arch is one implementation of the Arch SCM specification by Tom Lord.
- Nick Moffitt provides an interesting perspective on Arch in "Revision Control with Arch: Introduction to Arch" (Linux Journal, November 2004).
- Learn more about Git from Linus himself in "Torvalds Gives Inside Skinny on Git" (eWeek, April 2005).
- Explore Git in this Kernel Hackers' Guide.
- Check out Aegis, a transaction-based SCM, at sourceforge.
- With IBM trial software, available for download directly from developerWorks, build your next development project on Linux.

© Copyright IBM Corporation 2006
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)

Version control for Linux