About Version Control Systems

Version control systems are important to research and teaching. We compare the centralized and distributed models.

Why use version control?

1. Protection

Version control is like having infinite undo. Imagine that you are working on some code. You get it working, but then make a few additional changes. You return to the code later that day to find that you broke it. Version control makes it easy to see what has changed, and possibly revert your code back to a known state.

You release some code, and a month later someone finds a critical bug. In the intervening month your code has diverged substantially from your previous release, making it difficult to fix what may be a simple problem. Version control makes it easy to return to a previous state of your code in order to fix a problem without losing all of your intervening work.

Why version control?

2. Isolation

Version control makes it easy to experiment with your code. You can test out experimental features in a dedicated branch of your project without affecting the production code (or the work of your collaborators).

You can merge your changes into another branch or discard them when you have finished experimenting.

3. Collaboration

Version control makes it easier for multiple developers to work on the same project.

It makes it easier to share code.

A VCS handles the task of generating patches and merging changes, and makes it easier for a group of people to keep local working trees in sync.

It promotes accountability.

In a large project with many contributors, a version control system keeps track of who has made which changes. If a problem crops up, it's easy to identify the person responsible for the change.

Version control vocabulary

- **Repository**: A database containing the files and change history of your project.
- **Working tree or Working copy**: A local copy of files from a repository.
- **Vocabulary**
- **Revision**: The state of the repository at a certain point in time.
- **Commit**: To save your changes back to the repository.
- **Merge**: To combine two sets of changes to the files in your project.
- **Tag**: Identifies a point-in-time snapshot of your project.
- **Branch**: An isolated stream of changes to your project.

History of version control

In the beginning (c. 1985), there was RCS.

uses locking to manage conflicts.

But...

managed files, not projects.

everyone worked in the same place.

locks were inconvenient.

History of version control
RCS begat CVS.

uses merging and conflict detection to manage conflicts.
supports distributed operation.
But...
operations were not atomic
no support for renaming files/directories
History of version control

CVS begat Subversion.
designed to address problems in CVS
command line familiar to CVS users.
atomic operations, handles directories and renames.
The Subversion documentation describes the development of Subversion thusly:

In early 2000, CollabNet, Inc. (http://www.collab.net) began seeking developers to write a replacement for CVS... CVS's limitations were obvious from the beginning, and CollabNet knew it would eventually have to find something better... So CollabNet determined to write a new version control system from scratch, retaining the basic ideas of CVS, but without the bugs and misfeatures.

History of version control

The version control explosion.

git
Bazaar
darc
Mercurial
monotone
git and mercurial

Git and Mercurial both stem directly from the brouhaha surrounding the adoption – and subsequent rejection in 2005 – of the commercial BitKeeper version control system for the Linux kernel.

Bazaar

Bazaar was developed in 2005 by Canonical as a replacement for baz, which was itself a fork of GNU arch.

Darcs

Darcs was developed in 2002 as a result of the author's experience with Gnu arch.

Monotone

Monotone was initially released in 2003. In 2005 it was briefly a candidate for replacing BitKeeper for use in Linux kernel development.

Centralized version control

CVS, Subversion
One main repository
Commits go to central repository
Centralized version control

Developers check out working copies.
Centralized version control

Someone commits bad code to repository.
Centralized version control

Changes are visible to everyone.
Distributed version control

Most recent version control systems use a distributed model.
Distributed version control

Developers check out working copies.
Distributed version control

Someone commits bad code to local repository.
Distributed version control

Fixes locally and pushes to remote repository.
Distributed version control

Everyone is happy.

Distributed version control

There is no spoon.

In the world of distributed version control, the idea of a central repository is a social construct rather than a technical one. While some projects may find it convenient to identify a central repository, git (and other DVC systems) do not enforce a hub and spoke configuration.

For some of my own projects I have something of an "inverted tree": my working copies push to two remote repositories. One is a "personal" repository, which I use to coordinate my work between my office, my laptop, and so forth. The other is a "public" repository, where I push my code when I want others to see it.

Centralized vs. Distributed

It may sound like I am suggesting that distributed version control is generally better than centralized version control.

I am.

There are other opinions.

In particular, some of the developers of Subversion have suggested that a distributed model makes it less likely that people will share code with others (while in a centralized system they are largely forced to if they want to take advantage of the version control system).