Choosing a Version Control System

About this white paper: This paper was written by David C. Young, an employee of CSRA. It was written as supplemental documentation for use by the HPC account holders at the Alabama Supercomputer Center (ASC). This was originally written in 2011, and updated in 2013 and again in 2017.

Why have version control?

Anyone who has put a large amount of work into software development would be upset if the source code files were lost due to a hardware failure or incorrectly typed command. Some computers have an automatic nightly backup, and others don't. Even when there is a nightly backup, it would be convenient to be able to go back to the version of a file as it looked a few hours ago when some new code didn't work as expected. This is a particularly important issue in software development because a given file may be modified many, many times as the software is written. Version control systems are designed to store, compare, and manage many versions of computer software source code files.

The manual option

One option is to frequently make copies of files that are being frequently modified. This can be done by copying the files to a directory, CD, flash drive, zip file, or tar file.

This method works, and has been used to manage some fairly complex projects such as the earliest versions of the Linux kernel. However, this is not a particularly convenient way of managing old versions of files. This method also becomes increasingly cumbersome as more programmers get involved in the project.

The lock-modify-unlock option

The first version control systems were central repositories of "good" files. Each software developer could make modifications to a working copy in their own account then put those changes into the repository. The problem of two people modifying the same file at once was solved by a lock-modify-unlock system. When a developer wanted to modify a file, they checked it out which put a lock on the file that prevented anyone else from editing the file. When they were done, they checked their new version into the central repository and the lock was removed.

This system works, but there are issues. People couldn't always do their work because someone else had a necessary file locked. People would forget they had files checked out when they went on vacation leading to internal turmoil, getting the repository manager to unlock the file, then the changes made by the absent employee were lost. In those days, programmers frequently walked around the cubical farm saying "Hey Joe, when can you get the user_config.c file checked back in?".

The copy-modify-merge system

The next generation of version control systems (i.e. CVS or Subversion) used a copy-modify-merge system. Under this system, each developer got a copy of the file. Two people could modify their copy of the same file, then both could merge the file back into the repository. If the two developers working on the same file modified different subroutines, the merge software could integrate their changes automatically. If the merge system could not automatically merge the files, a variety of command line and graphic interface tools would be available for helping them to manually compare, choose, and edit.

Centralized file repositories

These first two types of version control systems store their repositories in a central server. The advantage of this is that we know where the source code is and can take the appropriate measures to make that server reliable.

There are some disadvantages to having a central server. You need to have someone administer the server. If the server or network is down, programmers will be unable to do their work, or at least hindered by system lag. Anyone that has experienced this knows that speed counts. You can wait 2 seconds for a repository command to run without loosing your train of thought. However, if you wait 20 seconds, your mind will wander and you will go off to work on other things while you wait. Modern computers and networks are very powerful, but if you have developers on multiple continents it can take Herculean efforts to keep the server access lag under 2 seconds, and you still won't be completely successful.

Another concern is the frequency at which you commit changes to the repository. The project manager would prefer to have changes submitted infrequently when the code is complete and thoroughly tested, so that it doesn't break the nightly build and test cycle. Programmers would like to submit changes multiple times per hour so that they can easily undo any changes that don't work as intended. You resolve this issue in a central repository by making a branch (slightly different version of the code) then merging that branch back in when it is working correctly. Doing this may require having the repository administrator create the branch, or getting management approval.

Distributed version control systems

Newer version control systems like Mercurial and Git are distributed. Each programmer's working copy is actually a complete copy of the source code repository. This allows developers to make new branches at any time conveniently. It also gives significantly faster version control commands because you don't need to wait for network and server traffic. Very small projects can also do away with the technical administration of a central server and repository. However, most projects still use a server to maintain the main copy called the “master” or “trunk”. That server might be a public website like GitHub, SourceForge, or Bitbucket. Or it might be a private server only accessible to authorized employees. Having a locally stored repository also means that changes since your last backup can be lost if your hard drive dies.

In the simplest usage, you could use Mercurial just like you use Subversion and ignore the fact that it is distributed. It is more responsive than Subversion and the commands to update and merge are similar. Doing this with Git is possible, but a bit more onerous.

Distributed systems facilitate improved work habits. Because every working directory is actually a branched off copy of the repository you can use "branchy" workflows. For example, every programmer can have several branches in progress for bugs, experimental features, etc.

You can also easily maintain multiple versions of the software and copy code updates between them as often or as infrequently as desired. For example, you might have a main project named Linux which has versions called Redhat and Debian, and version Debian might have a derivative named Ubuntu, etc.

The push-pull trade off

Early central repository systems were "push" systems. This means that developers push their code changes into the repository when they are ready. This works well when you can trust all of the programmers to always do reasonable things and put good code into the repository.

There are also "pull" systems. Everyone has copies of the repository, which are branches and branches off of branches. Changes only move between these branches when someone pulls changes from someone else’s repository copy into their own. This works well for open source software projects where anyone can contribute to the project regardless of whether that person is an expert, beginner, or a hacker working for an organized crime syndicate. These projects work off of a network of trust. When a developer makes a change to the code, they then petition the person heading up the project or that little corner of the project to pull their changes. If the person writing the new change is a first time contributor, their changes may be examined closely and they may be asked to fix a few problems before petitioning for inclusion, called submitting a “pull request”. If they are known to have done a lot of good work on this section of the code, their changes may be accepted into the main branch without anyone glancing at them.

Git originally was a "pull" only system, but recent versions support "push" functionality also. Many projects managed with Git disable the "push" functionality and manage the project as a "pull" only system.

Mercurial has both "push" and "pull" functionality by default, although it is fairly easy to disable either if so desired. The default configuration makes a "pull" a bit more convenient to use than a "push", but that can be changed with a couple lines in a configuration file.

Project Workflows

Even once your team has decided on a version control system and a push or pull mechanism, there are multiple ways to organize the source code repository and development process. Here are a few of the most popular options.

A Centralized Workflow. This is how work is done in an old-style central repository system (i.e. CVS) where there is really only one main repository (called “master” in Git or called “trunk” in Subversion). People work in the master branch and usually push their changes to the central repository. This is what CVS and Subversion are designed to do, although Subversion now has branches. This is possible in Git or Mercurial, but awkward and not leveraging the capabilities of those programs. The problem here is still that developers like to commit changes frequently so they can be reverted, but committing them to a main trunk means that the trunk tip is frequently broken by half written code. A tag on the main trunk can indicate a major or minor version release. Today this this type of workflow is most often an intermediate step as a development team transitions from an older version control system to leveraging the capabilities of one of the newer ones.

To implement a centralized workflow in a distributed versioning system like Git, the repository is created as a “bare” repository. This means it does not have a working directory so no one can work directly in the main repository. People work in their own complete copies of the repository, in the master branch. When they are ready to put their new work into the main repository, they first rebase (add new changes into their own repository), resolve any conflicts created by rebasing, then push their rebased copy with all of their new work into main repository (or submit frequent pull requests).

A Feature Branch Workflow. In this model, every time a developer starts working on a new feature they create a new branch off of the master line (and hopefully name it well so everyone knows what it's for). The developer can do hundreds of commits of half written code to this branch to save their work. When they feel it is done and thoroughly tested, they submit a pull request. At this point other developers and project developers can review the code, ask the developer to make changes, perhaps even add a bit to it. Once everyone is happy with it, it must be merged into the master branch, either by a project director or by the developer. The intent of this workflow is that; first the master branch is never broken, second the master never contains half written code, and third the pull requests act as a conduit for dialogue between the members of the development team. A tag on the master branch can denote a version release. The Feature Branch Workflow works well for small development teams, but can become unwieldy for big projects.

The Gitflow Workflow. Larger teams with more complex projects may choose a Gitflow Workflow. This has a few more core branches.
* The master branch contains ONLY release versions all tagged with release numbers.
* A development branch is where new work is being committed when done.
* Feature branches come off of development and merge back into the development branch.
* Before a release is made, a release branch comes off of the development branch. This is a feature freeze point. Additions to the release branch are test and bug fixes only. When ready, the release branch is merged to master with its version number tag. The release branch may then be frozen and no more additions made to it.
* A hotfix branch from master and back on to master can be done for emergency fixes.
* Additional maintenance branches may be made if old versions are being supported.
There is some extra overhead to this process. Changes made in a release branch, maintenance branch, or hotfix branch needed to be merged into master, and into development, and possibly into maintenance branches for other versions. Sometimes bug fixes are done in the development branch then need to be merged backwards into maintenance branches. There is a risk that back porting bug fixes may

accidentally pulling features backwards with them or creating new bugs because the bug fix was designed to work with a refactored version of the code. There is also training and management overhead as developers must understand the branch setup, and managers must enforce it. In spite of these potential issues, the Gitflow workflow is popular because it can effectively manage very large projects.

The Fork Workflow. In the workflows discussed above, there is one official branch called master or trunk, and one official version of the repository. Typically, the server is set up to clearly identify this branch and visitors wanting to download the software but not develop it are directed to the appropriate branch or release. These visitors cannot see what each developer is working on. In a Fork workflow, each developer forks off a copy of the main repository (a server side clone). Now the public can see multiple main trunks, typically one per developer. The developers make their own local copy of the repository and branch as needed as though they were a bunch of one-person projects. The developers can push to their public server repository when desired. Typically one person is a maintainer who pulls changes from multiple repositories when other developers send pull requests, but not necessarily from all of the developers. All of the developers can pull from the maintainer or other developers as desired. Proponents claim that the Fork Workflow lets projects grow organically with little or no management. Opponents point out that the Fork Workflow can result in projects getting fragmented into multiple public releases being supported by groups of developers who disagree ideologically about the project direction. Members of the general public may choose not to utilize any of them in favor of a project that has a clearly defined and advertised stable release version.

A fork workflow may, upon first look at the repositories, look like a feature branch workflow managed through a public server. The difference is in the directions updates are being pulled. In a feature branch workflow, the developers only pull from the main repository, and only the integrator pulls from the developers (as described below). In a fork workflow, any developer may be pulling from any other developer, and there might be multiple integrators each selectively choosing which updates to pull based on the emphasis of the branch they are maintaining.

A variation on the Fork Workflow is having multiple forks, each managed with a Gitflow mechanism and only rarely pulling changes between them. For example, there may be complete projects for Debian, Redhat, and SUSE, which only infrequently merge in changes developed by the other projects. Similarly new open source products have been developed by starting with a fork off of an existing project then taking it in a new direction.

The Integrator Workflow. The integrator workflow is an extra add-on set of steps inside any of these other workflows. If all of the developers are working on a shared file system, directory permissions could be set to allow others to pull from each developer's local copy of the repository. However, in many cases the person managing the project can't pull from another developer's local copy because that local copy is on the developer's laptop. In this case, the integration is done via a main repository. This can be a web service like GitHub or BitBucket, or a server owned and managed by your corporation. In order for the person managing the project to pull changes from the developers, those updates must be on the public server. Thus each developer creates a public repository, which is a server side copy created with a clone or fork. Each developer then creates a private repository on their laptop from their public repository. The developer does work in their private repository. When they are satisfied with it, they push it to their public repository on the server. Then the developer issues a pull request to the project managers. One of the project managers now takes on the role of integrator. The integrator pulls changes from the developer's public repository to their private repository. This gives the integrator a chance to check out the change, which can be as simple as checking it merges cleanly if they have confidence in that developer, or could be a careful look at what code was changed, if it was documented, compiles, includes unit tests, etc. Sometimes the pull request acts as vehicle for public discussion on the work. Once the project managers are satisfied with the change, they push it to the public main repository. In this format, project managers have permissions to push to the main repository, but developers do not. This provides a mechanism where anyone can have a chance to contribute to a project, even though the project managers may not know who that person is, if they write good code, if they might try to manipulate the code to create a backdoor, etc.

In software development, particularly open source projects, developers come and go, sometimes without warning. If a developer disappears without tying up loose ends, you can still grab the last thing they pushed to their public repository, both the completed changes and feature branches. If the person who was acting as integrator disappears, a different developer can simply designate their public repository as the new main repository. In distributed version control systems, such as git, the version control system does not inherently define one repository as the main one. Thus designation of which is the main repository is purely a matter of how the developers organize their work. This is unlike a centralized version control system, like Subversion, where the developer is working on a copy of the current files, but old versions and project history are only stored in the central repository.

Related issues

There is a natural tendency to adapt tools to new uses, or use tools in ways that were never intended. Version control systems are no exception. Although version control systems are designed for storing versions of software source code, they can be used to store versions of other types of files. This works best when storing versions of text files, since all of the merge and comparison functionality is designed for text files.

Most version control systems can store binary files, but there is little or no way to compare versions of these files, and every version of a binary file is often stored in it's entirety which results in the repository taking up quite a bit of disk space. This is often a clunky solution, which would be more efficient if the files were instead stored on systems designed to be file servers or document archives.

Computer system administrators will often use version control systems to store versions of the text files that contain system configuration settings. This works perfectly well since these files are written with a defined syntax as is software source code.

There are document repositories where co-workers will store multiple versions of word processor documents, images, videos, presentations, etc. Some of these are essentially just shared directories. Others are more aware of the type of data and whether two files are older and newer versions of the same document. Although version control systems can be used for this task, they aren't ideally suited. It is advisable to explore solutions designed for this type of usage before deciding to manage such documents with a version control system.

There are software development portals. Software development portals combine a version control system with related functions such as bug databases, shared documentation, project management tools, and other collaboration tools. These are often public web sites, but may be private servers whether accessed through a web browser or other software. Some of the most popular software development portals are Bitbucket, GitHub, and SourceForge. There are many others, all with various functionality, cost, advantages, and disadvantages.

Comparing version control systems

The most popular version control systems are currently Subversion, Git, and Mercurial. There are many others, including commercial systems like Perforce or ClearCase that have strengths, weaknesses, and selling points. There are also many web sites and graphic interfaces that provide a version control system, which in turn uses one of these under the hood.

Subversion is stable and reliable. It lets organizations stay with their tried and true code management practices. Recent versions of Subversion have better branching and merging capabilities than the earlier versions.

Git is incredibly flexible and powerful. However, it is the most difficult to use. New documentation and graphic interfaces are making it easier to use, but it will always be a very complex tool under the hood. Many projects using Git are only utilizing a tiny percentage of its capabilities. Git is currently very onerous to get working under Windows.

Mercurial brings the advantages of a distributed version control system with ease of use and good documentation. It can be a bit onerous to install due to dependencies on other required packages. On any given installation, certain of Mercurial's extensions may not work or may generate errors that can usually be ignored or disabled. This isn’t usually a significant problem.

WARNING: Mercurial uses Python. This can be a major problem if you are working on a project that needs a Python version that is incompatible with your Mercurial installation.

Recommendations

Anyone writing software should have some plan as to how they will ensure the safety of their code... even if that is as simple as depending upon nightly backups. For any but the very smallest projects, a software version control system is beneficial. The people involved in any project should consider which version control system is best for their needs at the present and for the near future. Many of these systems have ways to import and export source code repositories in case you have a need to change systems in the future.

For further information

http://subversion.apache.org/

http://mercurial.selenic.com/

http://git-scm.com/

http://www.perforce.com/

http://www-03.ibm.com/software/products/us/en/clearcase/

https://bitbucket.org/

https://github.com/

http://sourceforge.net/

"Mercurial; The Definitive Guide" B. O'Sullivan, O'Reilly, 2009

"Version Control with Subversion" C. M. Pilato, B. Collins-Sussman, B. W. Fitzpatrick, O'Reilley, 2008

"Version Control with Git: Powerful Tools and Techniques for Collaborative Software Development" J. Loeliger, O'Reilley, 2009

https://www.atlassian.com/git/tutorials/comparing-workflows

http://nvie.com/posts/a-successful-git-branching-model/