Centralized vs. distributed version control systems

Martin Žatkulák

In the world of programming and engineering, it is necessary to keep track of changes. This allows developers and programmers revert unwanted applications changes in the development process.

Version Control Systems (VCS) are standalone applications that have made the job easier by providing systemized management of multiple revisions of the same unit of information, whether it is a simple document or a digital document, such as source code of application or blueprint of electronic model. VCS allow developers to work collaboratively on the same application from different locations using repositories.

1 Centralized and distributed nature

Git was designed from the ground up as a distributed version control system. Being a distributed version control system means that multiple redundant repositories and branching are first class concepts of the tool.

In a distributed VCS like *Git* every user has a complete copy of the repository data stored locally, thereby making access to file history extremely fast, as well as allowing full functionality when disconnected from the network. It also means every user has a complete backup of the repository. Have 20 users? You probably have more than 20 complete backups of the repository as some users tend to keep more than one repository for the same project. If any repository is lost due to system failure only the changes which were unique to that repository are lost. If users frequently push and fetch changes with each other this tends to be an incredibly small amount of loss, if any.

In a centralized VCS like *Subversion* only the central repository has the complete history. This means that users must communicate over the network with the central repository to obtain history about a file. It also means that having 20 users does not automatically imply 20 active backups. Backups must be maintained independently of the VCS. If the central repository is lost due to system failure it must be restored from backup and all changes since that last backup are likely to be lost. Depending on the backup policies in place this could be several man-weeks worth of work.

2 Repositories

Since *Subversion* only supports a single repository there is little doubt about where something is stored. Once a user knows the repository URL they can reasonably assume

that all materials and all branches related to that project are always available at that location. Backup to tape/CD/DVD is also simple as there is exactly one location that needs to be backed up regularly.

Since *Git* is distributed by nature not everything related to a project may be stored in the same location. Therefore there may be some degree of confusion about where to obtain a particular branch, unless repository location is always explicitly specified. There may also be some confusion about which repositories are backed up to tape/CD/DVD regularly, and which aren't.

3 Access control

Due to being distributed, you inherently do not have to give commit access to other people when using *Git*. Instead, you decide when to merge what from whom. (There exist different mechanisms of control in case you do want to have a repository into which multiple people can push to.)

Since *Subversion* has a single central repository it is possible to specify read and write access controls in a single location and have them be enforced across the entire project.

4 Branch handling and accountability

Branches in *Git* are a core concept used everyday by every user. In *Subversion* they are more cumbersome and used sparingly. Branching can still be done with *Subversion* and within a company or organization this might make sense because a manager could check progress of new features by looking at the commits on branches and merges to the trunk. With open source however, you don't care what others are doing or about tracking their progress, you are simply thankful for any contribution.

The reason branches are so core in *Git* is every developer's working directory is itself a branch. Even if two developers are modifying two different unrelated files at the same time it's easy to view these two different working directories as different branches stemming from the same common base revision of the project.

Consequently *Git*:

- **Automatically tracks the project revision the branch started from.** Knowing the starting point of a branch is necessary in order to successfully merge the branch back to the main trunk that it came from.
- **Automatically records branch merge events.** Merge records always include the following details:
  - Who performed the merge.
  - What branch(es) and revision(s) were merged. (All changes made on the branch(es) remain attributed to the original authors and the original timestamps of those changes.)

  - What additional changes were made to complete the merge successfully. (Any changes made during the merge that is beyond those made on the branch(es) being merged is attributed to the user performing the merge.)
  - When the merge was done.
  - Why the merge was done.

- **Automatically starts the next merge at the last merge.** Knowing what revision was last merged is necessary in order to successfully merge the same branches together again in the future.

This is quite contrary to *Subversion*'s handling of branches. As of *Subversion* 1.5:

- **Automatically tracks the project revision the branch started from.** Like *Git*, *Subversion* remembers where a branch originated.

- **Incomplete merge event record.** Although *Subversion* records a merge as a commit and thus associates a username and a timestamp to it (like *Git*) there are some serious flaws in this record:

  - All changes made on the branch appear to be made by the merging user. This means that from a historical perspective every line of code modified on the branch will appear in the trunk as though it was written by the user who merged the branch. This is wrong if there were other users working on that branch.
  - It's impossible to see only merge related changes. If the merging user had to modify 12 lines of code to complete the merge successfully you can't tell what those 12 lines were, or how those 12 lines differ from the versions on the branches being merged.

In *Subversion*, branches and tags all are copies, it's a smart idea, but sometimes it's not convenient, many newbies checkout the whole repository by mistake or are confused when updating or merging a moved branch. Branch path and file path lie in same namespace but they have different semantics indeed and should be taken care in different way.

5 Performance

Git is extremely fast. Since all operations (except for `push` and `fetch`) are local there is no network latency involved to:

- Perform a `diff`.
- View file history.
- Commit changes.

- Merge branches.
- Obtain any other revision of a file (not just the prior committed revision).
- Switch branches.

6 Space requirements

Git's repository and working directory sizes are extremely small when compared to *Subversion*.

For example the Mozilla repository is reported to be almost 12 GB when stored in *Subversion* using the FSFS backend. Previously, the FSFS backend also required over 240000 files in one directory to record all 240000 commits made over the 10 year project history. This was fixed in *Subversion* 1.5, where every 1000 revisions are placed in a separate directory. The exact same history is stored in *Git* by only two files totaling just over 420 MB. *Subversion* requires 30 times more disk space to store the same history.

An *Subversion* working directory always contains two copies of each file: one for the user to actually work with and another hidden in `.svn/` to aid operations such as status, `diff` and `commit`. In contrast a *Git* working directory requires only one small index file that stores about 100 bytes of data per tracked file. On projects with a large number of files this can be a substantial difference in the disk space required per working copy.

As a full *Git* clone is often smaller than a full checkout, this means that *Git* working directories (including the repositories) are typically smaller than the corresponding *Subversion* working directories. There are even ways in *Git* to share one repository across many working directories, but in contrast to *Subversion*, this requires the working directories to be colocalized.

7 Line ending conversion

Subversion can be easily configured to automatically convert line endings to `CRLF` or `LF`, depending on the native line ending used by the client's operating system. This conversion feature is useful when Windows and UNIX users are collaborating on the same set of source code. It is also possible to configure a fixed line ending independent of the native operating system. Files such as a Makefile need to only use `LF`s, even when they are accessed from Windows. This can be adjusted in a global config and overridden in user configs. Binary files are checked in with a binary flag (like with CVS except that *Subversion* does this almost always automatically) and such never get converted or keyword substituted. Although additionally *Subversion* allows the user to specify line ending conversion on a file-by-file basis. But if the user does not check binary flag on adding (*Subversion* prints for every added file whether it recognized it as binary) binary content might get corrupted.

Whilst *Git* versions prior 1.5.1 never convert files and always assume that every file is opaque and should not be modified. *Git* 1.5.1 and onwards make this configurable. For

users on Windows they should set `core.autocrlf = true` so that text files are automatically checked out with `CRLF` and checked in as `LF`. *Git*'s advantage over *Subversion* is that you do not have to manually specify which files this conversion should be applied to, it happens automatically (hence `autocrlf`).

8 User interfaces

Currently *Subversion* has a wider range of user interface tools than *Git*. For example there are *Subversion* plugins available for most popular IDEs. There is a Windows Explorer shell extension. There are a number of native Windows and Mac OS X GUI tools available in ready-to-install packages.

Git's primary user interface is through the command line. There are two graphical interfaces: *git-gui* (distributed with *Git*) and *qgit*, which is making great strides towards providing another feature-complete graphical interface. Also *gitk*, the graphical history browser, can be more than just a fancy log reader. *git-gui* and *gitk* usually work out-of-box for common operating systems, and *qgit* is being ported to Qt4, which improves its portability. There are some user interface tools in development for *Git*, namely *TortoiseGit*, a port of *TortoiseSVN*. There is also *Git Extensions*, another explorer shell extension.

9 Partial checkout

With *Subversion*, you can check out just a subdirectory of a repository. Such a thing is not possible with *Git*. For a large project, this means that you always have to download the whole repository, even if you only need the current version of some sub-directory. In times where fast Internet connections are only available in most cities and traffic over mobile internet connections is expensive, *Git* can cost much more time and money in rural areas or with mobile devices.

10 Revision numbering

First, as *Subversion* assigns revision numbers sequentially (starting from 1) even very old projects such as Mozilla have short unique revision numbers (Mozilla is only up to 6 digits in length). Many users find this convenient when entering revisions for historical research purposes. They also find this number easy to embed into their product, supposedly making it easy to determine which sources were used to create a particular executable. However since the revision number is global to the entire repository, including all branches, there is still a question of which branch the revision number corresponds to. (Unless the last committed revision is recorded. Since revisions are global for a repository, the last committed revision makes it possible to determine which branch was used)

As *Git* uses a SHA1 to uniquely identify a commit each specific revision can only be described by a 40 character hexadecimal string, however this string not only identifies the revision but also the branch it came from. In practice the first 8 characters tends to be unique for a project, however most users try to not rely on this over the long term. Rather than embedding long commit SHA1s into executables *Git* users generate a uniquely named tag. This is an additional step, but a simple one.

Secondly, *Subversion*'s revision numbers are predictable. If the current commit is 435 the next one will be 436. It's very easy then to go through a few sequential revisions to, e.g. look at differences, revert to an old revision to find when a regression was introduced, etc. Furthermore, without looking up any additional information, you know that commit 436 was done after 435. Similar actions and knowledge from *Git* requires looking at the log.

11 Conclusion

By looking at an overview of the features of *Git* and *Subversion*, we can see that *Git* is preferable in most circumstances. Developers, researchers, engineers and other users of VCS have more inclination towards *Git*, but generally it strongly depends on your personal needs. Neither is best, but one is often better for what you are doing.

References

[1] Azad, S., *SVN vs. Git: Who Will Be the Future of Revision Control?*, 2008, available at `http://www.richappsconsulting.com/blog/blog-detail/svn-vs-git-who-will-be-the-future-of-revision-control/`.

[2] Deceth, *Git vs SVN – Which is Better?*, 2009, available at `http://www.looble.com/git-vs-svn-which-is-better/`.

[3] Git Wiki contributors, *GitSvnComparison*, 2006–2010, available at `http://git.wiki.kernel.org/index.php/GitSvnComparsion`.

[4] The Git Community Book contributors, *The Git Community Book*, 2010, available at `http://book.git-scm.com/`.