Version control concepts and best practices

by Michael Ernst

September, 2012
Last updated: March 3, 2018

This document is a brief introduction to version control. After reading it, you will be prepared to perform simple tasks using a version control system, and to learn more from other documents that may lack a high-level coneptual overview. Most of its advice is applicable to all version control systems, but its examples use Git, Mercurial (Hg), and Subversion (SVN) for concreteness.

This document's main purpose is to lay out philosophy and advice that I haven't found elsewhere in one place. It is not an exhaustive reference to the syntax of particular commands. This document covers basics, but it does not go into advanced topics like branching and rebasing, nor does it discuss the ways in which large projects use version control differently than small ones.

Contents:

(Also see How to create and review a GitHub pull request.)

Introduction to version control

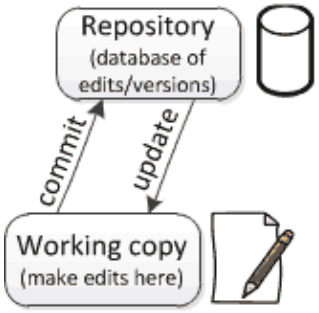
If you are already familiar with version control, you can skim or skip this section.

A version control system serves the following purposes, among others.

- Version control enables multiple people to simultaneously work on a single project. Each person edits his or her own copy of the files and chooses when to share those changes with the rest of the team. Thus, temporary or partial edits by one person do not interfere with another person's work.
  Version control also enables one person you to use multiple computers to work on a project, so it is valuable even if you are working by yourself.
- Version control integrates work done simultaneously by different team members. In most cases, edits to different files or even the same file can be combined without losing any work. In rare cases, when two people make *conflicting edits* to the same line of a file, then the version control system requests human assistance in deciding what to do.
- Version control gives access to historical versions of your project. This is insurance against computer crashes or data lossage. If you make a mistake, you can roll back to a previous version. You can reproduce and understand a bug report on a past version of your software. You can also undo specific edits without losing all the work that was done in the meanwhile. For any part of a file, you can determine when, why, and by whom it was ever edited.

Repositories and working copies

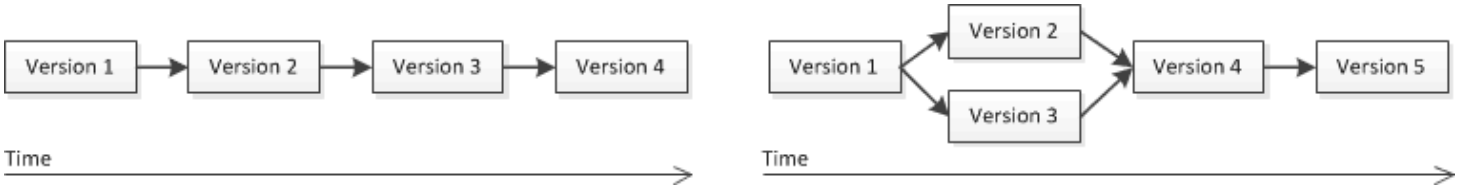
Version control uses a *repository* (a database of changes) and a *working copy* where you do your work.



Your *working copy* (sometimes called a *checkout*) is your personal copy of all the files in the project. You make arbitrary edits to this copy, without affecting your teammates. When you are happy with your edits, you `commit` your changes to a *repository*.

A repository is a database of all the edits to, and/or historical versions (snapshots) of, your project. It is possible for the repository to contain edits that have not yet been applied to your working copy. You can `update` your working copy to incorporate any new edits or versions that have been added to the repository since the last time you updated. See the diagram at the right.

In the simplest case, the database contains a linear history: each change is made after the previous one. Another possibility is that different users made edits simultaneously (this is sometimes called “branching”). In that case, the version history splits and then merges again. The picture below gives examples.

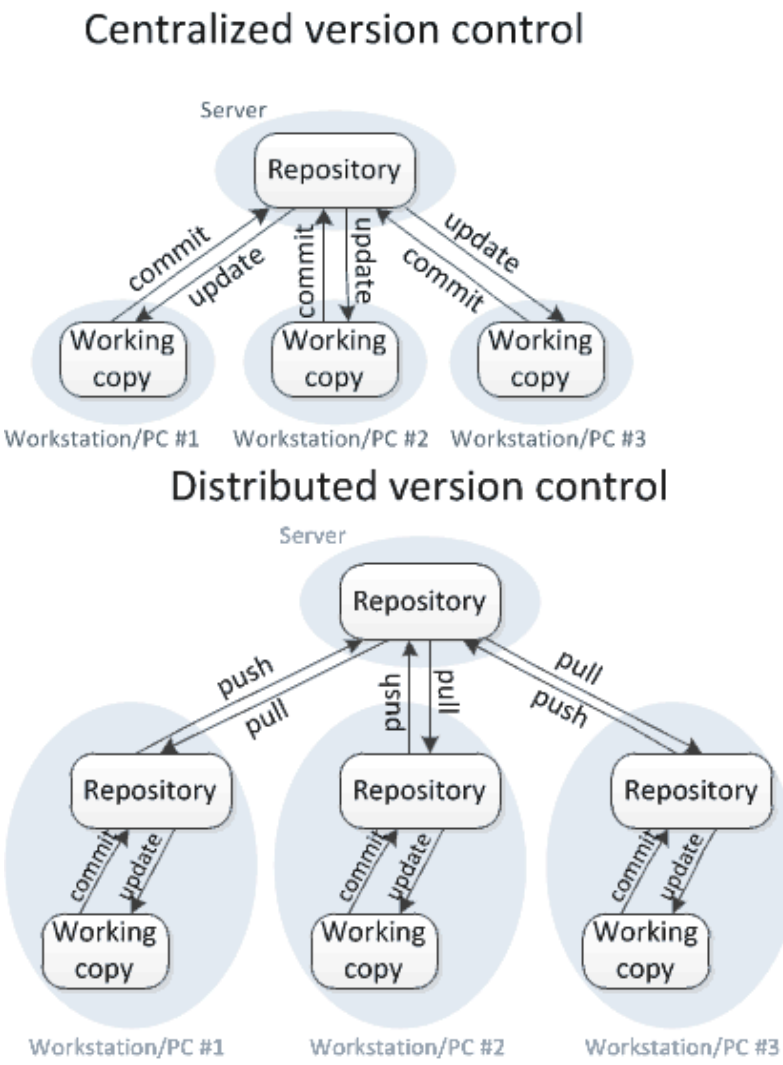


Distributed and centralized version control

There are two general varieties of version control: *centralized* and *distributed*. Distributed version control is more modern, runs faster, is less prone to errors, has more features, and is somewhat more complex to understand. You will need to decide whether the extra complexity is worthwhile for you.

Some popular version control systems are Git (distributed), Mercurial (distributed), and Subversion (centralized).

The main difference between centralized and distributed version control is the number of repositories. In centralized version control, there is just one repository, and in distributed version control, there are multiple repositories. Here are pictures of the typical arrangements:



In **centralized version control**, each user gets his or her own working copy, but there is just one central repository. As soon as you commit, it is possible for your co-workers to update and to see your changes. For others to see your changes, 2 things must happen:

- You commit
- They update

In **distributed version control**, each user gets his or her own repository *and* working copy. After you commit, others have no access to your changes until you push your changes to the central repository. When you update, you do not get others' changes unless you have first pulled those changes into your repository. For others to see your changes, 4 things must happen:

- You commit
- You push
- They pull
- They update

Notice that the commit and update commands only move changes between the working copy and the local repository, without affecting any other repository. By contrast, the push and pull commands move changes between the local repository and the central repository, without affecting your working copy.

It is sometimes convenient to perform both `pull` and `update`, to get all the latest changes from the central repository into your working copy. The `hg fetch` and `git pull` commands do both pull and update. (In other words, `git pull` does not follow the description above, and `git push` and `git pull` commands are not symmetric. `git push` is as above

and only affects repositories, but `git pull` is like `hg fetch`: it affects both repositories and the working copy, performs merges, etc.)

Conflicts

A version control system lets multiple users simultaneously edit their own copies of a project. Usually, the version control system is able to merge simultaneous changes by two different users: for each line, the final version is the original version if neither user edited it, or is the edited version if one of the users edited it. A *conflict* occurs when two different users make simultaneous, different changes to the same line of a file. In this case, the version control system cannot automatically decide which of the two edits to use (or a combination of them, or neither!). Manual intervention is required to resolve the conflict.

"Simultaneous" changes do not necessarily happen at the exact same moment of time. Change 1 and Change 2 are considered simultaneous if:

- User A makes Change 1 before User A does an update that brings Change 2 into User A's working copy, and
- User B makes Change 2 before User B does an update that brings Change 1 into User B's working copy.

In a distributed version control system, there is an explicit operation, called `merge`, that combines simultaneous edits by two different users. Sometimes `merge` completes automatically, but if there is a conflict, `merge` requests help from the user by running a merge tool. In centralized version control, merging happens implicitly every time you do `update`.

It is better to avoid a conflict than to resolve it later. The best practices below give ways to avoid conflicts, such as that teammates should frequently share their changes with one another.

Conflicts are bound to arise despite your best efforts. It's smart to practice conflict resolution ahead of time, rather than when you are frazzled by a conflict in a real project. You can do so in this tutorial about Git conflict resolution.

Merging changes

Recall that `update` changes the working copy by applying any edits that appear in the repository but have not yet been applied to the working copy.

In a centralized version control system, you can `update` (for example, `svn update`) at any moment, even if you have locally-uncommitted changes. The version control system merges your uncompleted changes in the working copy with the ones in the repository. This may force you to resolve conflicts. It also loses the exact set of edits you had made, since afterward you only have the combined version. The implicit merging that a centralized version control system performs when you `update` is a common source of confusion and mistakes.

In a distributed version control system, if you have uncommitted changes in your working copy, then you cannot run `update` (or other commands like `git pull` or `hg fetch` that themselves invoke `update`). The reason is that it would be confusing and error-prone for the version control system to try to apply edits, when you are in the middle of editing. You will receive an error message such as

```
abort: outstanding uncommitted changes
```

Before you are allowed to `update`, you must first `commit` any changes that you have made (you should continue editing until they are logically complete first, of course). Now, your repository database contains *simultaneous* edits — the ones you just made, and the ones that were already there and you were trying to apply to your working copy by running `update`. You need to `merge` these two sets of edits, then `commit` the result. (In Mercurial, you will typically just run `hg fetch`, which performs the `merge` and `commit` for you.) The reason you need the commit is that merging is an operation that gets recorded by the version control system, in order to record any choices that you made during merging. In this way, the version control system contains a complete history and clearly records the difference between you making your edits and you merging simultaneous work.

Version control best practices

The advice in this section applies to both centralized and distributed version control.

These best practices do not cover obscure or complex situations. Once you have mastered these practices, you can find more tips and tricks elsewhere on the Internet.

Use a descriptive commit message

It only takes a moment to write a good commit message. This is useful when someone is examining the change, because it indicates the purpose of the change. This is useful when someone is looking for changes related to a given concept, because they can search through the commit messages.

Make each commit a logical unit

Each commit should have a single purpose and should completely implement that purpose. This makes it easier to locate the changes related to some particular feature or bug fix, to see them all in one place, to undo them, to determine the changes that are responsible for buggy behavior, etc. The utility of the version control history is compromised if one commit contains code that serves multiple purposes, or if code for a particular purpose is spread across multiple different commits.

During the course of one task, you may notice another issue and want to fix it too. You may need to commit one file at a time — the `commit` command of every version control system supports this.

- Git: `git commit` *`file1 file2`* commits the two named files.
  Alternately, `git add` *`file1 file2`* "stages" the the two named files, causing them to be committed by the next `git commit` command that is run without any filename arguments.
- Mercurial: `hg commit` *`file1 file2`* commits the two named files, and `hg commit .` commits all the changed files in the current directory.
- Subversion: `svn commit` *`file1 file2`* commits the two named files, and `svn commit .` commits all the changed files in the current directory.

If a single file contains changes that serve multiple purposes, you may need to save your all your edits, then re-introduce them in logical chunks, committing as you go. Here is a low-tech way to do this; each version control system also has more sophisticated mechanisms to support this common operation.

- Git: Move *`myfile`* to a safe temporary location, then run `git checkout` *`myfile`* to restore *`myfile`* to its unmodified state (same as whatever is in the repository).
  Git contains more sophisticated ways to do this, such as staging some but not all of the changes in a given file to the index (also known as the cache), or stashing some of your changes. Once you are more comfortable with Git, you should learn about these mechanisms.
- Mercurial: `hg revert` *`myfile`* copies the current *`myfile`* to *`myfile`*`.orig` and restores *`myfile`* to its unmodified state (same as whatever is in the repository).
- Subversion: Move *`myfile`* to a safe temporary location, then run `svn update` *`myfile`* to restore *`myfile`* to its unmodified state (same as whatever is in the repository).

Sometimes it is too burdensome to separate every change into its own commit. However, aiming for (and often achieving) this goal will serve you well in the longer term.

Avoid indiscriminate commits

As a rule, I do not run `git commit -a` (or `hg commit` or `svn commit`) without supplying specific files to commit. If you supply no file names, then these commands commit every changed file. You may have changes you did not intend to make permanent (such as temporary debugging changes); even if not, this creates a single commit with multiple purposes.

When I want to commit my changes, to avoid accidentally committing more than I intended, I always run the following commands:

For Git:

```
# Lists all the modified files
git status
# Shows specific differences, helps me compose a commit message
git diff
# Commits just the files I want to
git commit file1 file2 -m "Descriptive commit message"
```

For Mercurial:

```
# Lists all the modified files
hg status
# Shows specific differences, helps me compose a commit message
hg diff
# Commits just the files I want to
hg commit file1 file2 -m "Descriptive commit message"
```

Incorporate others' changes frequently

Work with the most up-to-date version of the files as possible. That means that you should run `git pull`, `git pull -r`, `hg fetch`, or `svn update` very frequently. I do this every day, on each of over 100 projects that I am involved with.

When two people make conflicting edits simultaneously, then manual intervention is required to resolve the conflict. But if someone else has already completed a change before you even start to edit, it is a huge waste of time to create, then manually resolve, conflicts. You would have avoided the conflicts if your working copy had already contained the other person's changes before you started to edit.

Share your changes frequently

Once you have committed the changes for a complete, logical unit of work, you should share those changes with your colleagues as soon as possible (by doing `git push` or `hg push`). So long as your changes do not destabilize the system, do not hold the changes locally while you make unrelated changes. The reason is the same as the reason for incorporating others' changes frequently.

This advice is slightly different for centralized version control such as Subversion. This advice translates to running `svn commit`, which both commits and shares your changes, as often as possible. However, be careful because you cannot make private commits that do not affect your teammates.

Coordinate with your co-workers

The version control system can often merge changes that different people made simultaneously. However, when two people edit the same line, then this is a *conflict* that a person must manually resolve. To avoid this tedious, error-prone work, you should strive to avoid conflicts.

If you plan to make significant changes to (a part of) a file that others may be editing, coordinate with them so that one of you can finish work (commit and push it) before the other gets started. This is the best way to avoid conflicts. A special case of this is any change that touches many files (or parts of them), which requires you to coordinate with all your teammates.

Remember that the tools are line-based

Version control tools record changes and determine conflicts on a line-by-line basis. The following advice applies to editing marked-up text (LaTeX, HTML, etc.). It does not apply when editing WYSIWYG text (such as a plain text file), in which the intended reader sees the original source file.

Never refill/rejustify paragraphs. Doing so changes every line of the paragraph. This makes it hard to determine, later, what part of the content changed in a given commit. It also makes it hard for others to determine which commits

affected given content (as opposed to just reformatting it). If you follow this advice and do not refill/rejustify the text, then the LaTeX/HTML source might look a little bit funny, with some short lines in the middle of paragraphs. But, no one sees that except when editing the source, and the version control information is more important.

Do not write excessively long lines; as a general rule, keep each line to 80 characters. The more characters are on a line, the larger the chance that multiple edits will fall on the same line and thus will conflict. Also, the more characters, the harder it is to determine the exact changes when viewing the version control history. As another benefit to authors of the document, 80-character lines are also easier to read when viewing/editing the source file.

Don't commit generated files

Version control is intended for files that people edit. Generated files should not be committed to version control. For example, do not commit binary files that result from compilation, such as `.o` files or `.class` files. Also do not commit `.pdf` files that are generated from a text formatting application; as a rule, you should only commit the source files from which the `.pdf` files are generated.

- Generated files are not necessary in version control; each user can re-generate them (typically by running a build program such as `make` or `ant`).
- Generated files are prone to conflicts. They may contain a timestamp or in some other way depend on the system configuration. It is a waste of human time to resolve such uninteresting conflicts.
- Generated files can bloat the version control history (the size of the database that is stored in the repository). A small change to a source file may result in a rather different generated file. Eventually, this affects performance of the version control system.
  This is a particular problem when the generated file is binary. Version control systems can concisely record the differences between two versions of a textual file (usually the differences are much smaller than the file itself). However, version control systems have to store each version of a binary file in its entirety.

To tell your version control system to ignore given files, create a top-level `.gitignore` or `.hgignore` file, or set the `svn:ignore` property.

Understand your merge tool

The least pleasant part of working with version control is resolving conflicts. If you follow best practices, you will have to resolve conflicts relatively rarely.

You are most likely to create conflicts at a time you are stressed out, such as near a deadline. You do not have time, and are not in a good mental state, to learn a merge tool. So, you should make sure that you understand your merge tool ahead of time. When an actual conflict comes up, you don't want to be thrown into an unfamiliar UI and make mistakes. Practice on a temporary repository to give yourself confidence.

A version control system lets you choose from a variety of merge programs (example: Mercurial merge programs). Select the one you like best. If you don't want an interactive program to be run, you can configure Mercurial to attempt the merge and write a file with conflict markers if the merge is not successful.

Obtaining your copy

Obtaining your own working copy of the project is called "cloning" or "checking out":

- `git clone` *`URL`*
- `hg clone` *`URL`*
- `svn checkout` *`URL`*

Use your version control's documentation to learn how to create a new repository (`hg init`, `git init`, or `svnadmin create`).

Distributed version control best practices

Typical workflow

The typical workflow when using Git (or Mercurial) is:

- `git pull` (or `hg fetch`)
- As many times as desired (but usually very few times):
  - Make local edits
  - Examine the local edits: `git status` and `git diff` (or `hg status` and `hg diff`
  - `git commit` (or `hg commit`)
  - `git pull` or `git pull -r` (or `hg commit`)
- `git pull` or `git pull -r` (or `hg commit`)
- `git push` (or `hg push`)

Note that an invocation of `git pull` or `hg fetch` may force you to resolve a conflict.

That's pretty much all you need to know, besides how to clone an existing repository.

In Mercurial, use `hg fetch`, not `hg pull`

(This tip is specific to Mercurial. In Git, just use `git pull`. Git's `pull` acts similarly to Mercurial's `fetch`.)

I never run `hg pull`. Instead, I use `hg fetch`. It is the most effective way to get everyone else's changes into my working copy. The `hg fetch` command is like `hg pull` then `hg update`, but it is even more useful. The actual effect of `hg fetch` is:

- `hg pull`
- If merging is necessary:
  - `hg merge`
  - `hg commit`
- `hg update`

To enable the `hg fetch` command, add the following to your `$HOME/.hgrc` file or equivalent:

```
[extensions]
fetch =
```

There is nothing after the `=` in `fetch =`.

Don't force it

Git or Mercurial occasionally refuses to do a particular action, such as pushing to a remote repository when you have not yet pulled all its changes. For example, Mercurial indicates this problem by outputting:

```
abort: push creates new remote heads!
(did you forget to merge? use push -f to force)
```

Mercurial suggests that you merge the changes — the best way to do this is with `hg fetch` (not `hg merge`, then you can try again to push. Mercurial also notes that you can perform the operation (even though it is not recommended) by using the `-f` or `--force` command-line option. **Never** use `-f` or `--force`: doing so is likely to cause extra work for your team, such as making multiple people perform a merge.

Merging when you have uncommitted changes

As explained above, you cannot `update` until you `commit` and `merge`. You will see an error message like

```
abort: outstanding uncommitted changes
```

But, sometimes you really want to incorporate others' changes even though your changes are not yet in a logically consistent state and ready to commit to your local repository.

A low-tech solution is to revert your changes with `hg revert` or the analogous command for other version control systems, as described above. Now, you can `git pull` or `hg fetch`, but you will have to manually re-do the changes that you moved aside. There are other, more sophisticated ways to do this as well (for Mecurial, see the Mercurial FAQ; for Git, use `git stash`).

More tips

Caching your password

SVN (Subversion) automatically caches your password. You have to type the password only the first time.

Here are two ways to have Mercurial remember/cache your password so you don't have to type it every time.

1. `hg clone https://michael.ernst:`*`my-password-here`*`@jsr308-langtools.googlecode.com/hg/ jsr308-langtools`
2. In your global `.hgrc` file, add this section:

```
# The below only works in Mercurial 1.3 and later
[auth]

googlecode.prefix = code.google.com
googlecode.username = michael.ernst
googlecode.password = my-password-here
googlecode.schemes = https

dada.prefix = dada.cs.washington.edu/hgweb/
dada.username = mernst
dada.password = my-password-here
dada.schemes = https
```

Email notification

It's a good idea to set up email notification. Then, every time someone pushes (in distributed version control) or commits (in centralized version control) all the relevant parties get an email about the changes to the central server.

If you are using a hosted service such as GitHub or Bitbucket, it's easy to set up email notification on their website.

For a Mercurial repository you are hosting, to set up email notification, see https://www.mercurial-scm.org/wiki/NotifyExtension with a tutorial at http://morecode.wordpress.com/2007/08/03/setting-up-mercurial-to-e-mail-on-a-commit/ and a concrete example at https://github.com/mernst/uwisdom/blob/wiki/HgNotifyExtension.adoc.

The diffs in Mercurial's email notifications can be confusing. When sending one message per push (that is, when using the `changegroup.notify` setting), the diff in the email shows all the differences in all the changesets that you pushed. However, some of these changesets might be merge changesets resulting from `hg merge` or `hg fetch`. The changes in a merge changeset were already seen by the mailing list when the original author pushed his/her changes, and combining them all together obscures the new changes that appear for the first time in this push (which is, to a first approximation, everything but merges).
To solve this problem, configure the repository's `hgrc` file as follows:

```
[hooks]
# One email per changeset/commit, not one email per push
incoming.notify = python:hgext.notify.hook
[notify]
# Don't send notifications for merge changesets
merge = False
```

Back to Advice compiled by Michael Ernst.

Michael Ernst