**VERSION CONTROL REVIEW**

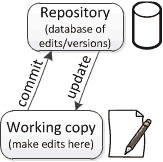
**VERIFACTS SERVICES PRIVATE LIMITED**

# Introduction to version control

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 us to use multiple computers to work on a project, so it is valuable even if we are working by ourself.
* 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](https://homes.cs.washington.edu/~mernst/advice/version-control.html#conflicts) 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 our project. This is insurance against computer crashes or data loss. If we make a mistake, we can roll back to a previous version. We can reproduce and understand a bug report on a past version of our software. We can also undo specific edits without losing all the work that was done in the meanwhile. For any part of a file, we can determine when, why, and by whom it was ever edited.

**Repositories and working copies**

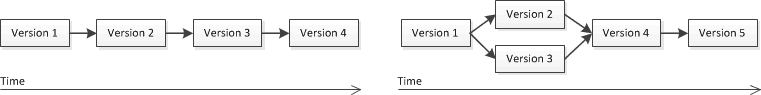


Version control uses a repository (a database of changes) and a working copy where we do our work.

Our working copy (sometimes called a checkout) is our personal copy of all the files in the project. We make arbitrary edits to this copy, without affecting our teammates. When we are happy with our edits, we commit our changes to a *repository*.

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

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.

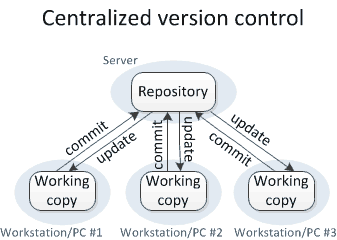
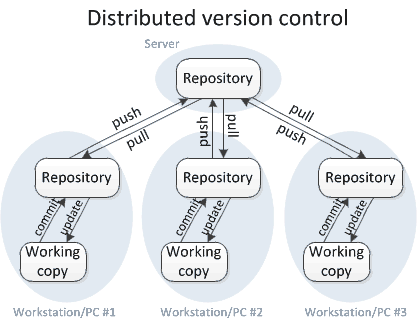


**Distributed and centralized version control**

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. We will need to decide whether the extra complexity is worthwhile for us.

Some popular version control systems are Git (distributed), Mercurial (distributed), and Subversion (centralized). In practice, almost all open-source projects use Git.

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 we commit, it is possible for our co-workers to update and to see our changes. For others to see our changes, 2 things must happen:

* We commit
* They update

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

* We commit
* We 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 our working copy.

It is sometimes convenient to perform both pull and update, to get all the latest changes from the central repository into our 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](https://homes.cs.washington.edu/~mernst/advice/version-control.html#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](https://homes.cs.washington.edu/~mernst/advice/version-control.html#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 we do update.

It is better to avoid a conflict than to resolve it later. The [best practices](https://homes.cs.washington.edu/~mernst/advice/version-control.html#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 our best efforts. It's smart to practice conflict resolution ahead of time, rather than when we are frazzled by a conflict in a real project.

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

In a distributed version control system, if we have uncommitted changes in our working copy, then we 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 we are in the middle of editing. We will receive an error message such as

Abort: outstanding uncommitted changes

Before we are allowed to update, we must first commit any changes that we have made (we should continue editing until they are [logically complete](https://homes.cs.washington.edu/~mernst/advice/version-control.html#logical-unit) first, of course). Now, our repository database contains *simultaneous* edits — the ones we just made, and the ones that were already there and we were trying to apply to our working copy by running update. We need to merge these two sets of edits, then commit the result. (In Mercurial, we will typically just run hg fetch, which performs the merge and commit for us.) The reason we need the commit is that merging is an operation that gets recorded by the version control system, in order to record any choices that we made during merging. In this way, the version control system contains a complete history and clearly records the difference between us making our edits and us 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 we have mastered these practices, we 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, we may notice another issue and want to fix it too. We 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 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, we may need to save our all our edits, then re-introduce them in logical chunks, committing as we 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 our changes. Once we are more comfortable with Git, we 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 us well in the longer term.

### Avoid indiscriminate commits

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

When we want to commit our changes, to avoid accidentally committing more than we intended, we 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 we should run git pull, git pull -r, hg fetch, or svn update very frequently. We do this every day, on each of over 100 projects that we are 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 us even start to edit, it is a huge waste of time to create, then manually resolve, conflicts. We would have avoided the conflicts if our working copy had already contained the other person's changes before we started to edit.

### Share our changes frequently

Once we have committed the changes for a complete, logical unit of work, we should share those changes with our colleagues as soon as possible (by doing git push or hg push). So long as our changes do not destabilize the system, do not hold the changes locally while we make unrelated changes. The reason is the same as the reason for [incorporating others' changes frequently](https://homes.cs.washington.edu/~mernst/advice/version-control.html#incorporate-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 our changes, as often as possible. However, be careful because we cannot make private commits that do not affect our teammates.

**Coordinate with our 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](https://homes.cs.washington.edu/~mernst/advice/version-control.html#conflicts) that a person must manually resolve. To avoid this tedious, error-prone work, we should strive to avoid conflicts.

If we plan to make significant changes to (a part of) a file that others may be editing, coordinate with them so that one of us 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 us to coordinate with all our 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/justify 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 we follow this advice and do not refill/justify 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, we 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 our 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 we follow best practices, we will have to resolve conflicts relatively rarely.

We are most likely to create conflicts at a time we are stressed out, such as near a deadline. We do not have time, and are not in a good mental state, to learn a merge tool. So, we should make sure that we understand our merge tool ahead of time. When an actual conflict comes up, we 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 us choose from a variety of merge programs (to see Git's list, run git mergetool --tool-help). Select the one we like best. If we don't want an interactive program to be run, we can configure our version control system to attempt the merge and write a file with conflict markers if the merge is not successful.

### Obtaining your copy

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

* git clone URL
* hg clone URL
* svn checkout URL

Use our 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 is:

* git pull
* As many times as desired:
  + Make local edits
  + Examine the local edits: git status and git diff
  + git commit
  + git pull or git pull -r
* git pull or git pull -r
* git push

Note that an invocation of git pull may force us to resolve a conflict.