Cal State Fullerton

CPSC 254

Software Development With Open Source Systems

- Version Control

Instructor: Tejaas Mukunda Reddy

Version Control

- What is “version control”, and why should you care? Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later. For the examples in this book, you will use software source code as the files being version controlled, though in reality you can do this with nearly any type of file on a computer.
- If you are a graphic or web designer and want to keep every version of an image or layout (which you would most certainly want to), a Version Control System (VCS) is a very wise thing to use. It allows you to revert selected files back to a previous state, revert the entire project back to a previous state, compare changes over time, see who last modified something that might be causing a problem, who introduced an issue and when, and more. Using a VCS also generally means that if you screw things up or lose files, you can easily recover. In addition, you get all this for very little overhead.
- A component of software configuration management, version control, also known as revision control or source control,[1] is the management of changes to documents, computer programs, large web sites, and other collections of information. Changes are usually identified by a number or letter code, termed the "revision number", "revision level", or simply "revision". For example, an initial set of files is "revision 1". When the first change is made, the resulting set is "revision 2", and so on. Each revision is associated with a timestamp and the person making the change. Revisions can be compared, restored, and with some types of files, merged.
- The need for a logical way to organize and control revisions has existed for almost as long as writing has existed, but revision control became much more important, and complicated when the era of computing began. The numbering of book editions and of specification revisions are examples that date back to the print-only era. Today, the most capable (as well as complex) revision control systems are those used in software development, where a team of people may change the same files.
- Version control systems (VCS) most commonly run as stand-alone applications, but revision control is also embedded in various types of software such as word processors and spreadsheets, collaborative web docs[2] and in various content management systems, e.g., Wikipedia's page history. Revision control allows for the ability to revert a document to a previous revision, which is critical for allowing editors to track each other's edits, correct mistakes, and defend against vandalism and spamming. -https://en.wikipedia.org/wiki/Version_control

- In computer software engineering, revision control is any kind of practice that tracks and provides control over changes to source code. Software developers sometimes use revision control software to maintain documentation and configuration files as well as source code.
- As teams design, develop and deploy software, it is common for multiple versions of the same software to be deployed in different sites and for the software's developers to be working simultaneously on updates. Bugs or features of the software are often only present in certain versions (because of the fixing of some problems and the introduction of others as the program develops).
- Therefore, for the purposes of locating and fixing bugs, it is vitally important to be able to retrieve and run different versions of the software to determine in which version(s) the problem occurs. It may also be necessary to develop two versions of the software concurrently: for instance, where one version has bugs fixed, but no new features (branch), while the other version is where new features are worked on (trunk).
- At the simplest level, developers could simply retain multiple copies of the different versions of the program, and label them appropriately. This simple approach has been used in many large software projects. While this method can work, it is inefficient as many near-identical copies of the program have to be maintained. This requires a lot of self-discipline on the part of developers and often leads to mistakes. Since the code base is the same, it also requires granting read-write-execute permission to a set of developers, and this adds the pressure of someone managing permissions so that the code base is not compromised, which adds more complexity. Consequently, systems to automate some or all of the revision control process have been developed. This ensures that the majority of management of version control steps is hidden behind the scenes.
- Moreover, in software development, legal and business practice and other environments, it has become increasingly common for a single document or snippet of code to be edited by a team, the members of which may be geographically dispersed and may pursue different and even contrary interests. Sophisticated revision control that tracks and accounts for ownership of changes to documents and code may be extremely helpful or even indispensable in such situations.
- Revision control may also track changes to configuration files, such as those typically stored in /etc or /usr/local/etc on Unix systems. This gives system administrators another way to easily track changes made and a way to roll back to earlier versions should the need arise.

Revision Control

- Revision control manages changes to a set of data over time. These changes can be structured in various ways.
- Often the data is thought of as a collection of many individual items, such as files or documents, and changes to individual files are tracked. This accords with intuitions about separate files but causes problems when identity changes, such as during renaming, splitting or merging of files. Accordingly, some systems, such as git, instead consider changes to the data as a whole, which is less intuitive for simple changes but simplifies more complex changes.
- When data that is under revision control is modified, after being retrieved by checking out, this is not in general immediately reflected in the revision control system (in the repository), but must instead be checked in or committed. A copy outside revision control is known as a "working copy". As a simple example, when editing a computer file, the data stored in memory by the editing program is the working copy, which is committed by saving. Concretely, one may print out a document, edit it by hand, and only later manually input the changes into a computer and save it. For source code control, the working copy is instead a copy of all files in a particular revision, generally stored locally on the developer's computer;[note 1] in this case saving the file only changes the working copy, and checking into the repository is a separate step.
- If multiple people are working on a single data set or document, they are implicitly creating branches of the data (in their working copies), and thus issues of merging arise, as discussed below. For simple collaborative document editing, this can be prevented by using file locking or simply avoiding working on the same document that someone else is working on.
- Revision control systems are often centralized, with a single authoritative data store, the repository, and check-outs and check-ins done with reference to this central repository. Alternatively, in distributed revision control, no single repository is authoritative, and data can be checked out and checked into any repository. When checking into a different repository, this is interpreted as a merge or patch.

Source-management models

- Traditional revision control systems use a centralized model where all the revision control functions take place on a shared server. If two developers try to change the same file at the same time, without some method of managing access the developers may end up overwriting each other's work. Centralized revision control systems solve this problem in one of two different "source management models": file locking and version merging.

File Locking

- The simplest method of preventing "concurrent access" problems involves locking files so that only one developer at a time has write access to the central "repository" copies of those files. Once one developer "checks out" a file, others can read that file, but no one else may change that file until that developer "checks in" the updated version (or cancels the checkout).
- File locking has both merits and drawbacks. It can provide some protection against difficult merge conflicts when a user is making radical changes to many sections of a large file (or group of files). However, if the files are left exclusively locked for too long, other developers may be tempted to bypass the revision control software and change the files locally, leading to more serious problems.

Version Merging

- Main article: Merge (revision control)
- Most version control systems allow multiple developers to edit the same file at the same time. The first developer to "check in" changes to the central repository always succeeds. The system may provide facilities to merge further changes into the central repository, and preserve the changes from the first developer when other developers check in.
- Merging two files can be a very delicate operation, and usually possible only if the data structure is simple, as in text files. The result of a merge of two image files might not result in an image file at all. The second developer checking in the code will need to take care with the merge, to make sure that the changes are compatible and that the merge operation does not introduce its own logic errors within the files. These problems limit the availability of automatic or semi-automatic merge operations mainly to simple text-based documents, unless a specific merge plugin is available for the file types.
- The concept of a reserved edit can provide an optional means to explicitly lock a file for exclusive write access, even when a merging capability exists.

Version Control Best Practices

Commit Related Changes

- A commit should be a wrapper for related changes. For example, fixing two different bugs should produce two separate commits. Small commits make it easier for other team members to understand the changes and roll them back if something went wrong. With tools like the staging area and the ability to stage only parts of a file, Git makes it easy to create very granular commits.

Commit Often

- Committing often keeps your commits small and, again, helps you commit only related changes. Moreover, it allows you to share your code more frequently with others. That way it's easier for everyone to integrate changes regularly and avoid having merge conflicts. Having few large commits and sharing them rarely, in contrast, makes it hard both to solve conflicts and to comprehend what happened.

Don't Commit Half-Done Work

- You should only commit code when it's completed. This doesn't mean you have to complete a whole, large feature before committing. Quite the contrary: split the feature's implementation into logical chunks and remember to commit early and often. But don't commit just to have something in the repository before leaving the office at the end of the day. If you're tempted to commit just because you need a clean working copy (to check out a branch, pull in changes, etc.) consider using Git's "Stash" feature instead.

Test Before You Commit

- Resist the temptation to commit something that you "think" is completed. Test it thoroughly to make sure it really is completed and has no side effects (as far as one can tell). While committing half-baked things in your local repository only requires you to forgive yourself, having your code tested is even more important when it comes to pushing / sharing your code with others.

Version Control Best Practices

Write Good Commit Messages

- Begin your message with a short summary of your changes (up to 50 characters as a guideline). Separate it from the following body by including a blank line. The body of your message should provide detailed answers to the following questions: What was the motivation for the change? How does it differ from the previous implementation? Use the imperative, present tense („change", not „changed" or „changes") to be consistent with generated messages from commands like git merge.

Version Control is not a Backup System

- Having your files backed up on a remote server is a nice side effect of having a version control system. But you should not use your VCS like it was a backup system. When doing version control, you should pay attention to committing semantically (see "related changes") – you shouldn't just cram in files.

Use Branches

- Branching is one of Git's most powerful features – and this is not by accident: quick and easy branching was a central requirement from day one. Branches are the perfect tool to help you avoid mixing up different lines of development. You should use branches extensively in your development workflows: for new features, bug fixes, experiments, ideas…

Agree on a Workflow

- Git lets you pick from a lot of different workflows: long-running branches, topic branches, merge or rebase, git-flow… Which one you choose depends on a couple of factors: your project, your overall development and deployment workflows and (maybe most importantly) on your and your teammates' personal preferences. However you choose to work, just make sure to agree on a common workflow that everyone follows.